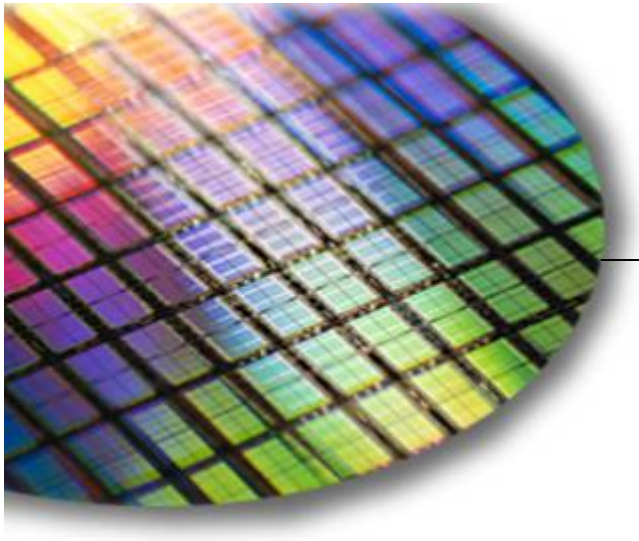


The World Leader in High Performance Signal Processing Solutions

Blackfin Overview

Srinivas K
A. Patil
A. Awasthy
Shailendra Miglani

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Agenda

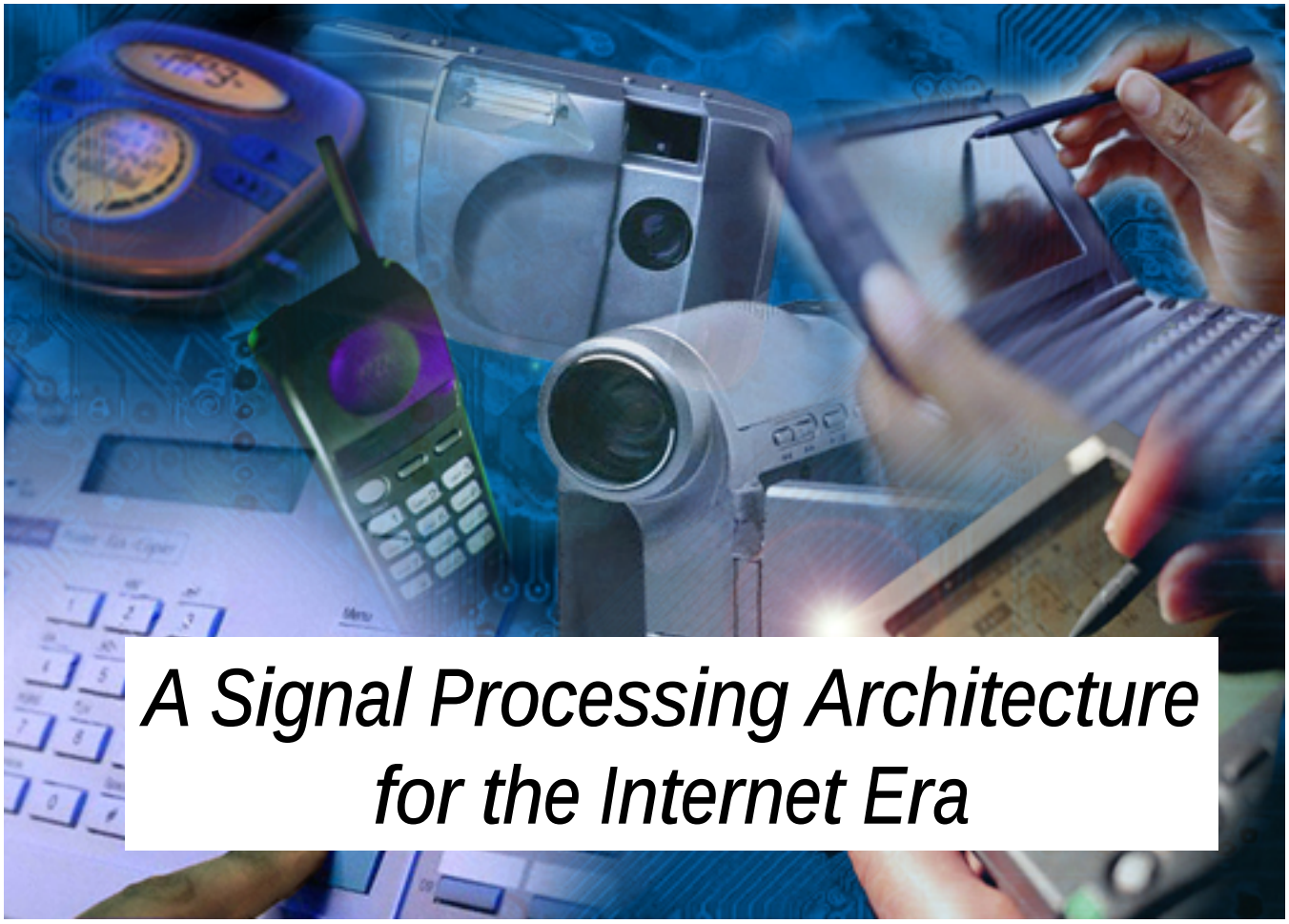
- **Day 1**
  - Introduction
  - VisualDSP++ features
  - Coding guidelines for achieving Optimal C Performance on Blackfin
  - Architecture and Pipeline
  - Memory
  - Assembly level optimization

- **Day 2**
  - Introduction to LDF
  - DMA
  - VDK and uClinux
  - Q & A session

—Analog Devices Confidential Information—

ANALOG DEVICES

Blackfin DSP Technology



—Analog Devices Confidential Information—

ANALOG DEVICES

Blackfin : Introduction

- **Blackfin DSP is the architectural base for a whole new family of DSPs from ADI. It is built upon the *Micro Signal Architecture (MSA)* core developed through the Joint Development with Intel Corporation.**
- **Blackfin DSPs incorporate the industry's highest performance 16-bit DSP architecture. It has Dynamic Power Management capabilities which delivers the lowest power consumption.**
- **Blackfin DSPs are optimized for processing data, communications and video streams for penetration into new market spaces.**

Blackfin : Features and Benefits

- **High Performance for real time video signal processing**
- **Easily programmed to support complex, new standards.**
- **Handles the DSP and Control code with equal efficiency.**
- **Maximizes work and minimizes energy per cycle**

- **High Performance**
  - **Blackfin offers 600M MACs today with a roadmap for 2G MACs**
- **Low Power Consumption**
  - **Blackfin DSP enables significant power savings by dynamically varying both voltage and frequency.**
- **Ease to use**
  - **Blackfin DSP combines attributes of both high performance DSP and microcontrollers into a single RISC ISA.**

BLACKfin Processors Embed MCU Features

- **Arbitrary bit and bit-field manipulation, insertion and extraction**
- **Integer operations on 8/16/32-bit data-types**
- **Memory protection and separate user and supervisor stack pointers**
- **Scratch SRAM for context switching**
- **Population and leading digit counting**
- **Byte addressing DAGs**
- **Compact Code Density**

Integrated Blackfin Features Typically Found in a Microcontroller

A RISC Instruction Set

Data Movement
LD, ST, 8,16,32 bits
Unsigned, Sign-extend
Register moves, P-D-DAG,
Push, Pop, Push/Popmult
CC2 dreg, etc.

Addressing Modes
Auto incr, Auto decr,
Pre-decr store on SP,
Indirect
Indexed w/immed offset
Post-incr w/ nonunity stride
Byte addressable

Program Control
BRCC, UJUMP,
Call, Rets, Loop Setup

Arithmetic
+,-,*,/,>>>, Negate
2 and 3 operand instructs

Logical
AND, OR, XOR, NOT
BITtst,set,tgl,clr, CC ops
<<,>>

Video
SAA, Byteops: Residual calc,
Spatial Interpolation, Spatial
Filter

Cache Control
Prefetch, Flush

and ...

Memory management

Event control

Supervisor/user modes

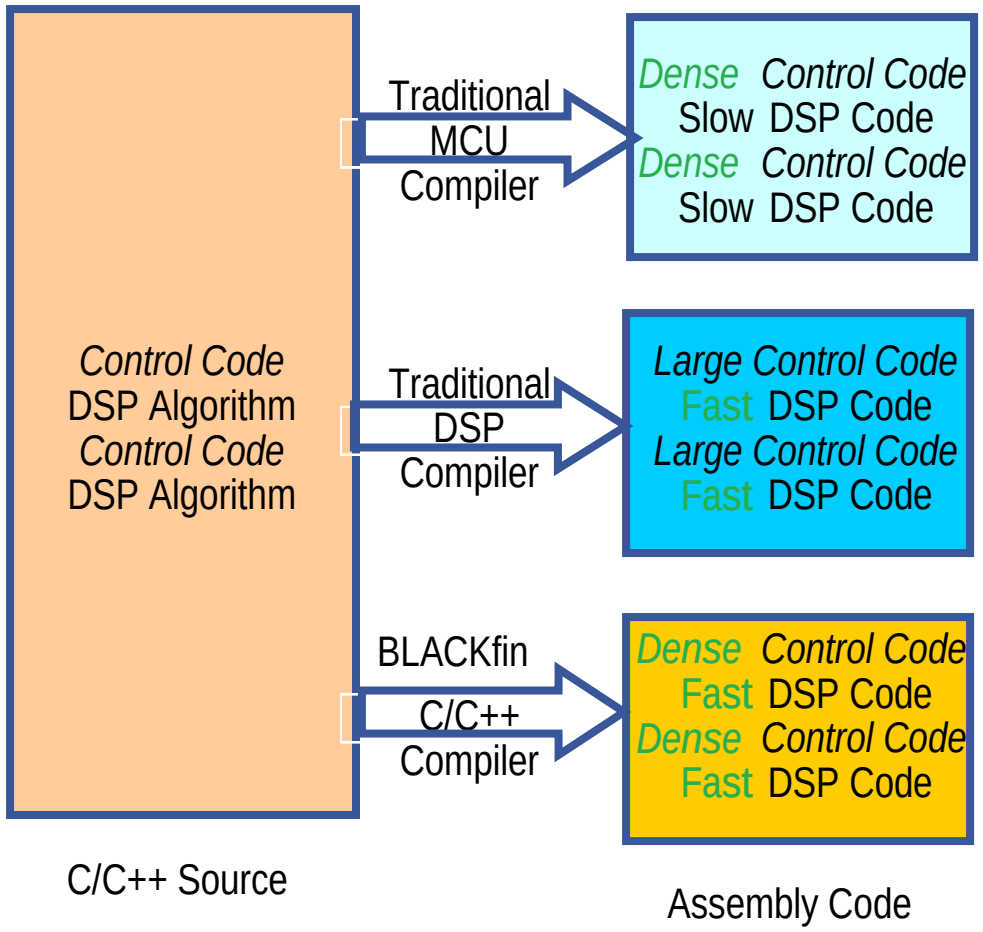
Wide range of peripherals

There is not a separate Micro-Controller mode!

—Analog Devices Confidential Information—

ANALOG DEVICES

BLACKfin Processors - Simplify Programmer's Model



Traditional MCU Compiler generates

- Dense control code, BUT
- Much larger and slower DSP code

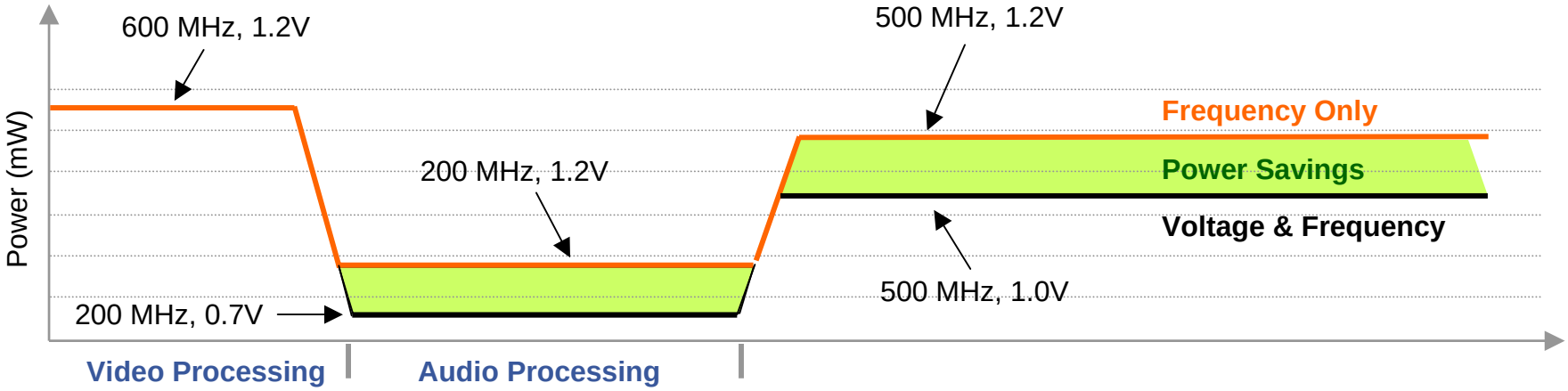
Traditional DSP Compiler generates

- Good DSP algorithm code, BUT
- Much larger control code

Architecture and Compiler Work Together to Deliver Dense Control Code and Fast DSP Code

ANALOG DEVICES

Enhanced Dynamic Power Management Increases Battery Life



- **Variable Frequency**
  - Programmable PLL (1x to 63x) combined with CCLK and SCLK dividers enable low latency changes in system performance and power consumption profile
- **Variable Voltage**
  - On-Chip Voltage Regulator generates core voltage from an externally supplied 2.25 – 3.6V input
  - Core voltage programmable from 0.7V to 1.2V (50 mV increments)
- **System Cost Reduction**

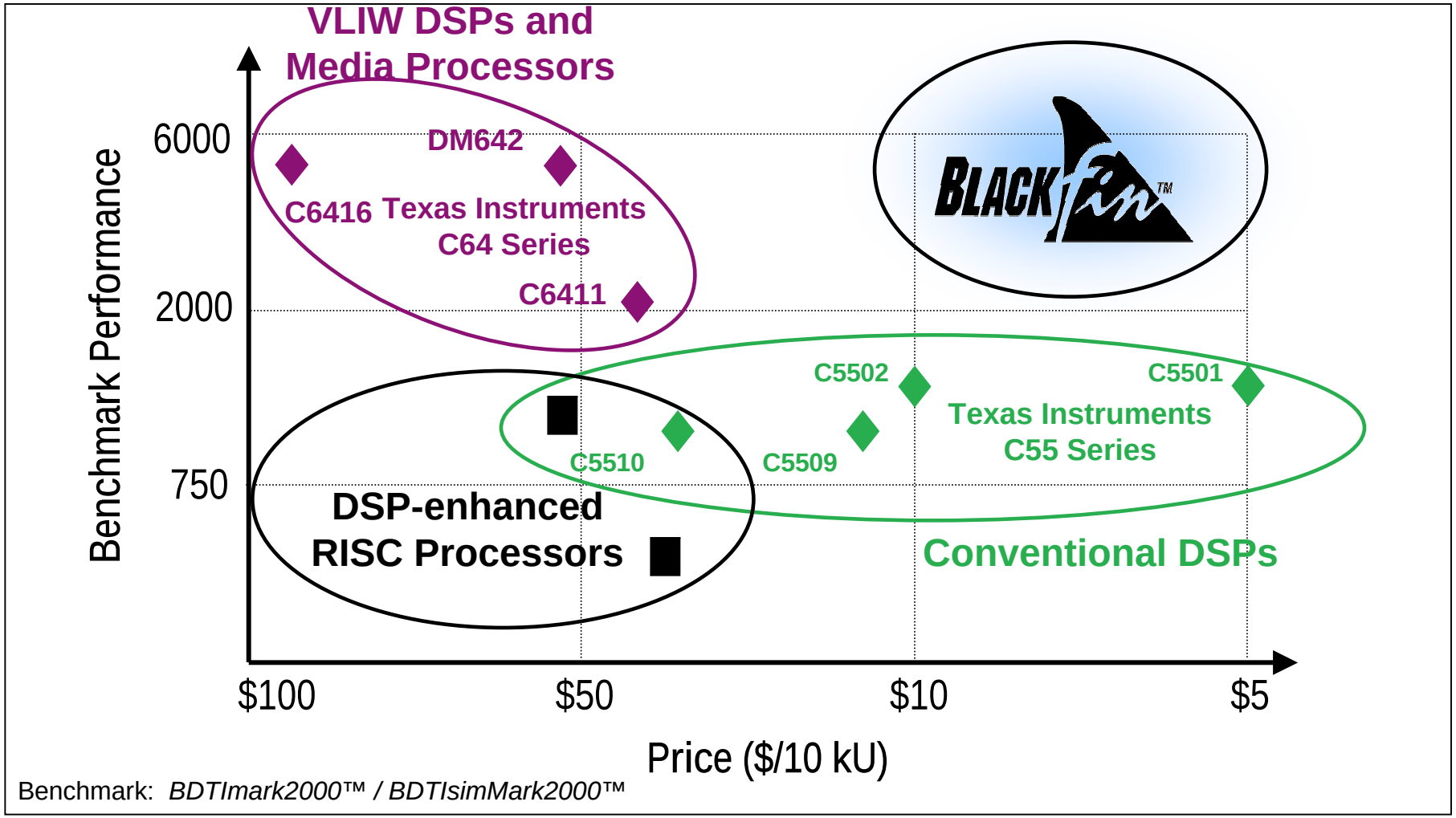
Blackfin : Target Applications

- **PDA**
  - Internet audio
- **Digital Still Camera**
- **Video camera**
  - Video conferencing
  - MPEG2
  - DVD
- **Digital Printing**
- **Audio**
  - MP3 Audio
  - Digital Car Radios

- **Modems**
  - ADSL
  - VoIP Phone Solutions
  - Cable Modems
  - RAS Modems
  - Wireless modems
- **Mobile Phones**
  - GSM Mobile phones
  - 3G data terminals
- **Internet Appliances**

—Analog Devices Confidential Information—

ADI Blackfin: Performance Leadership

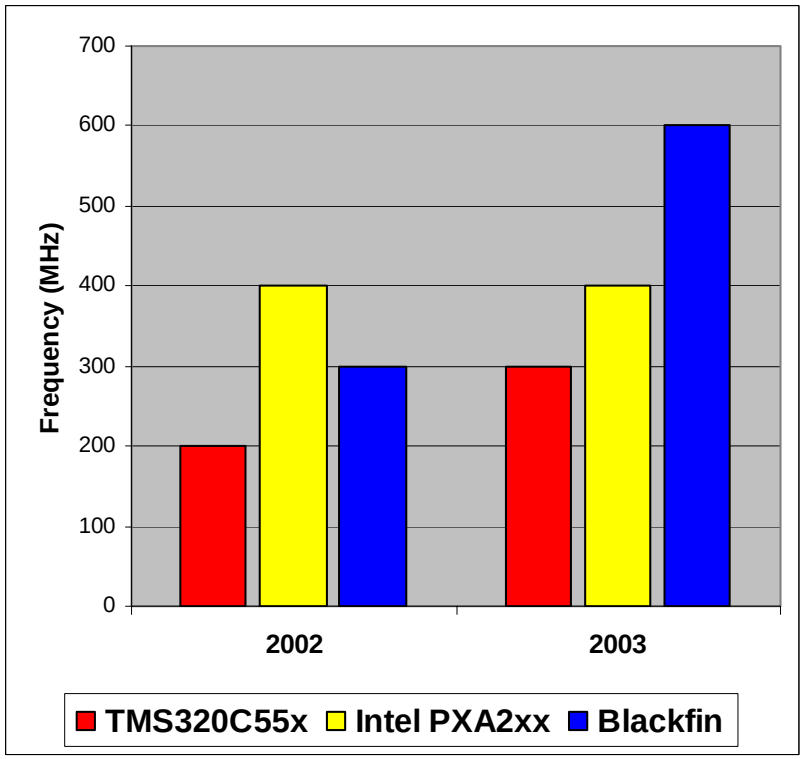


Benchmark: *BDTImark2000™ / BDTIsimMark2000™*

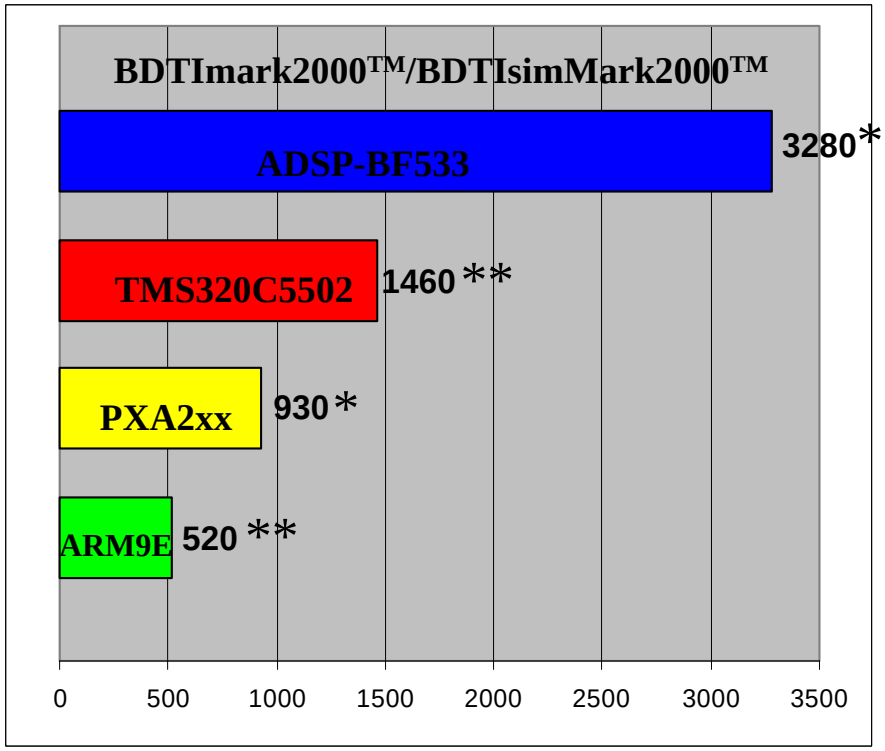
ANALOG DEVICES

Blackfin Competitive Performance Advantage

Blackfin has Higher Clock Rate ….



And > 2x Signal Processing Performance



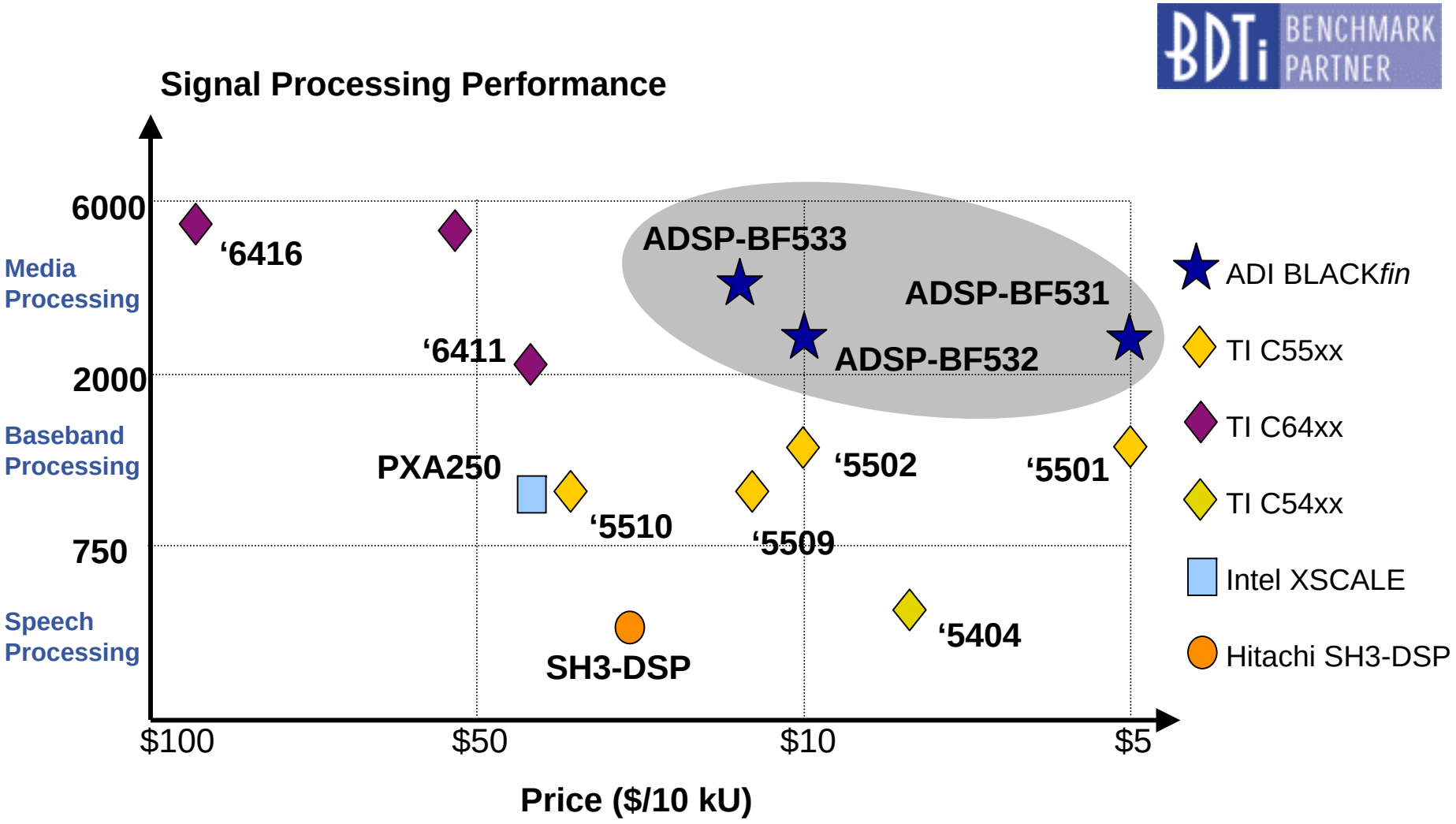
BDTi BENCHMARK PARTNER

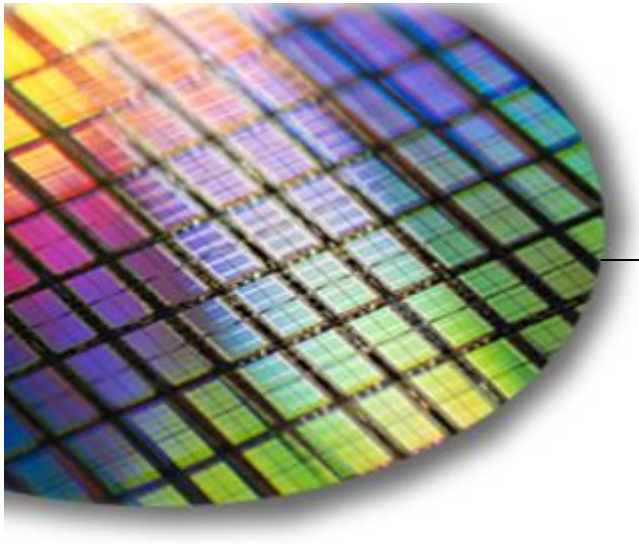
The BDTImark2000/BDTIsimMark2000 provide a summary measure of DSP speed. For more info and scores see www.BDTI.com. Scores © 2002/2003 BDTI.

*BDTImark2000 **BDTIsimMark2000 (simulated only, not verified on hardware)

ANALOG DEVICES

Price/Performance Comparison





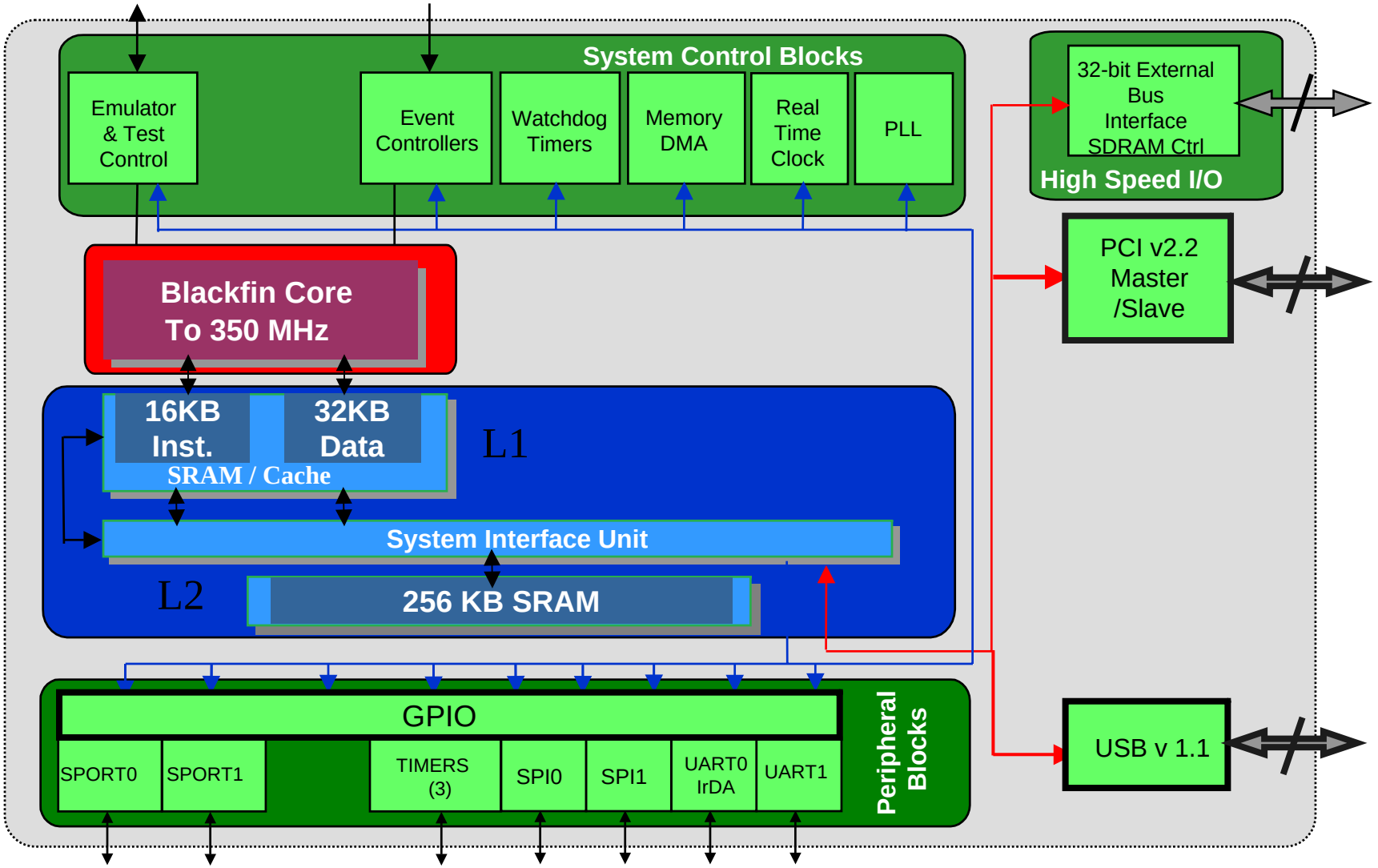
The World Leader in High Performance Signal Processing Solutions

Blackfin Products at a Glance

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

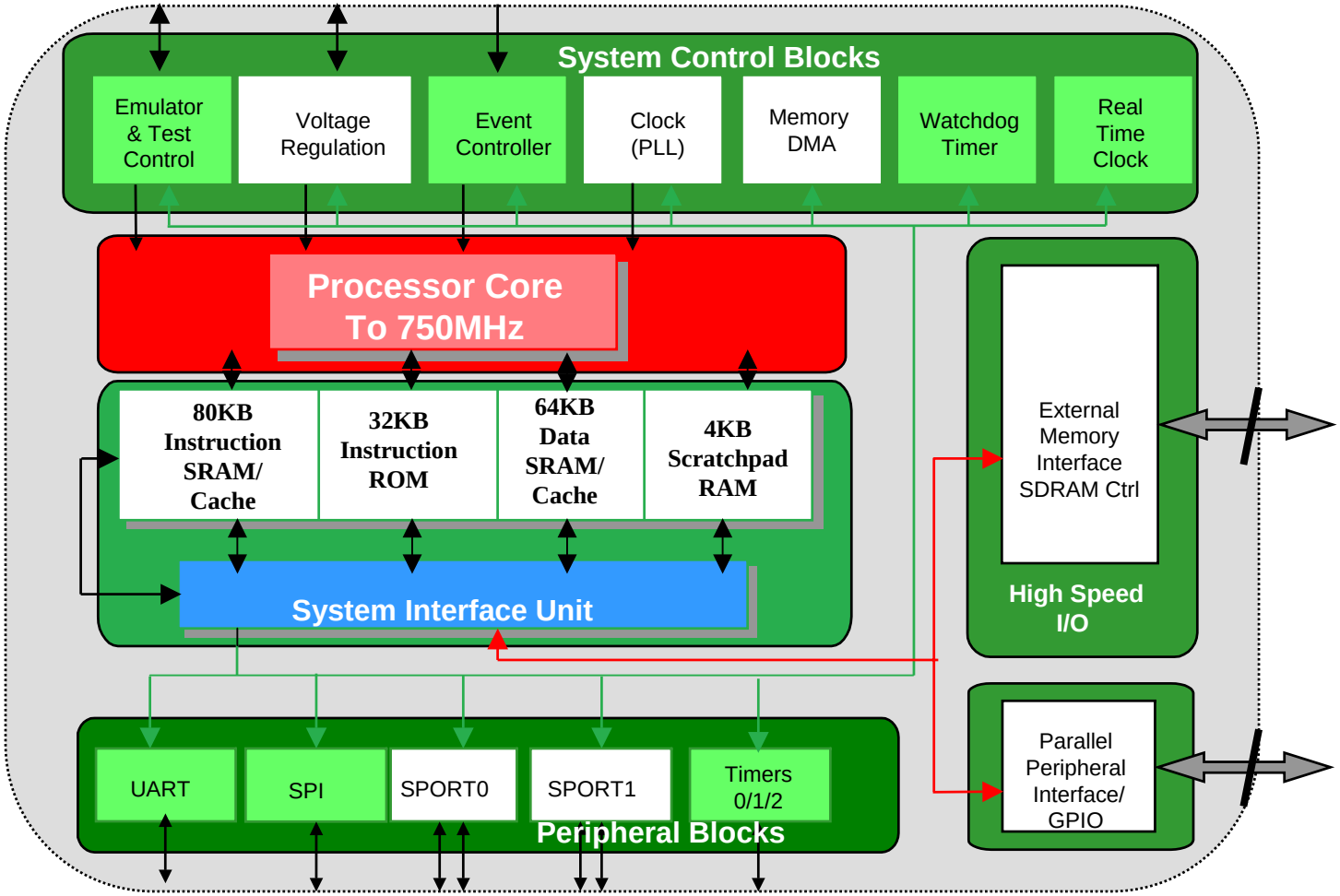
ADSP-BF535 Blackfin DSP – Available Now



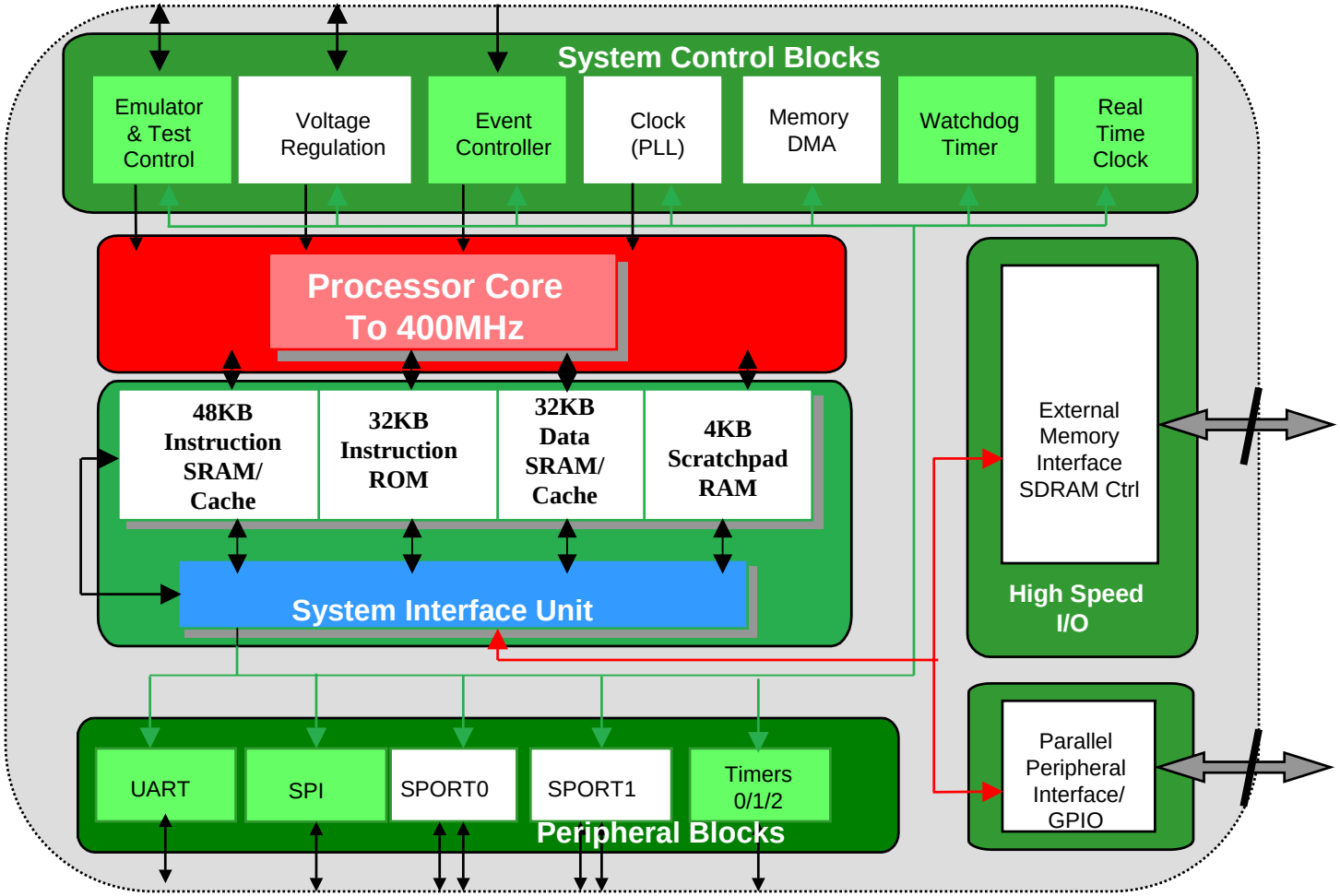
—Analog Devices Confidential Information—

ANALOG DEVICES

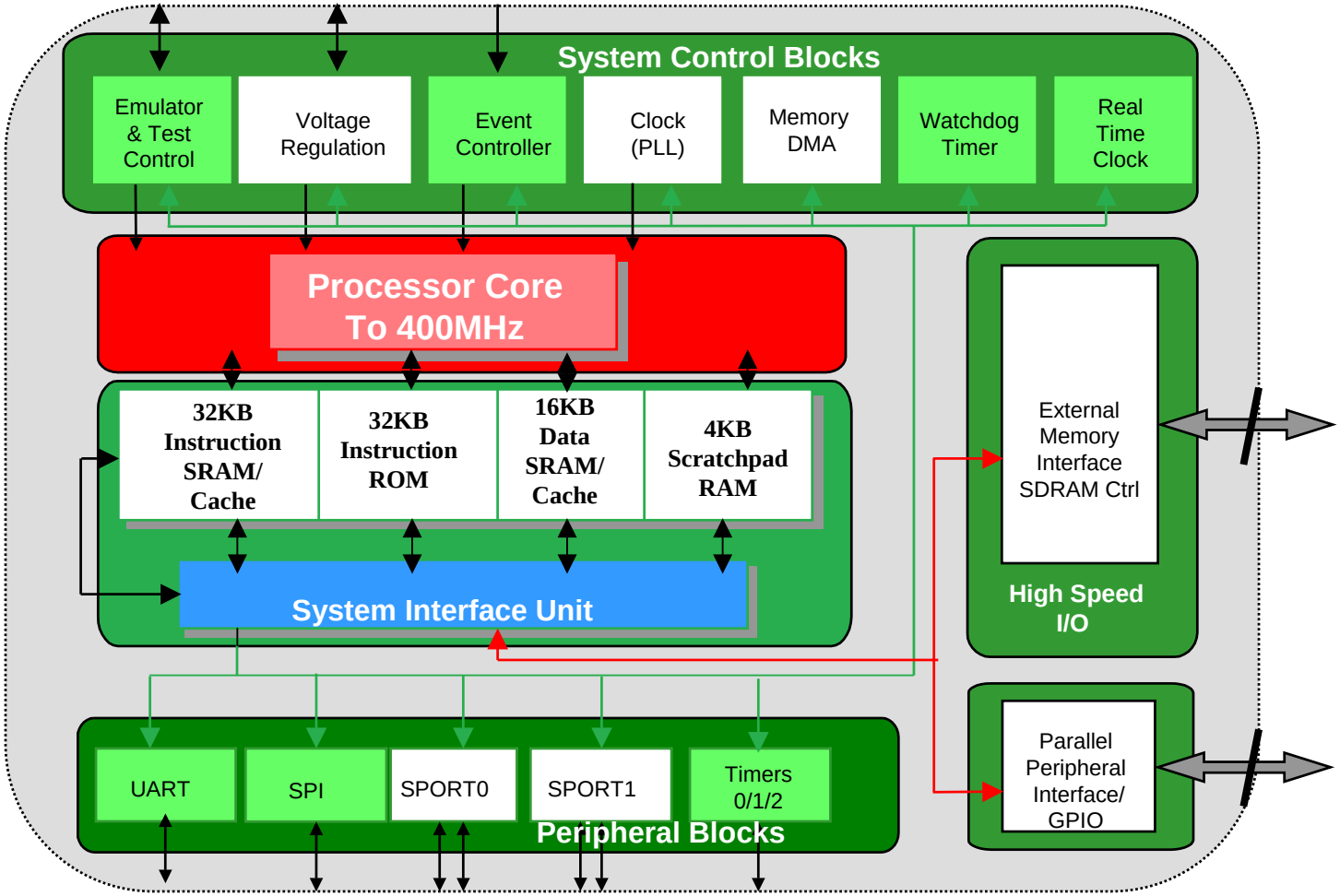
Blackfin : ADSP-BF533 – Available Now



Blackfin : ADSP-BF532 – Available Now

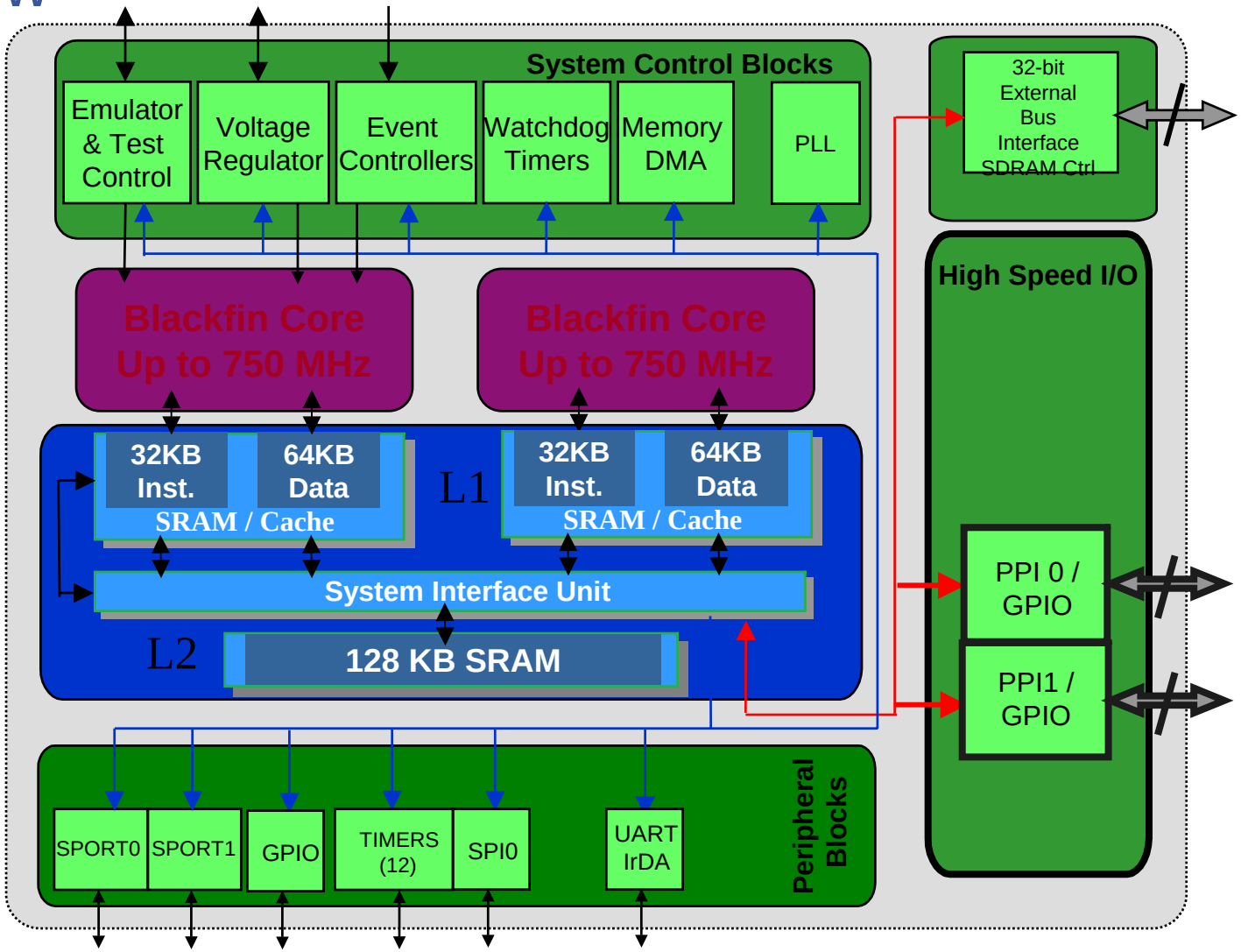


Blackfin : ADSP-BF531 – Available Now

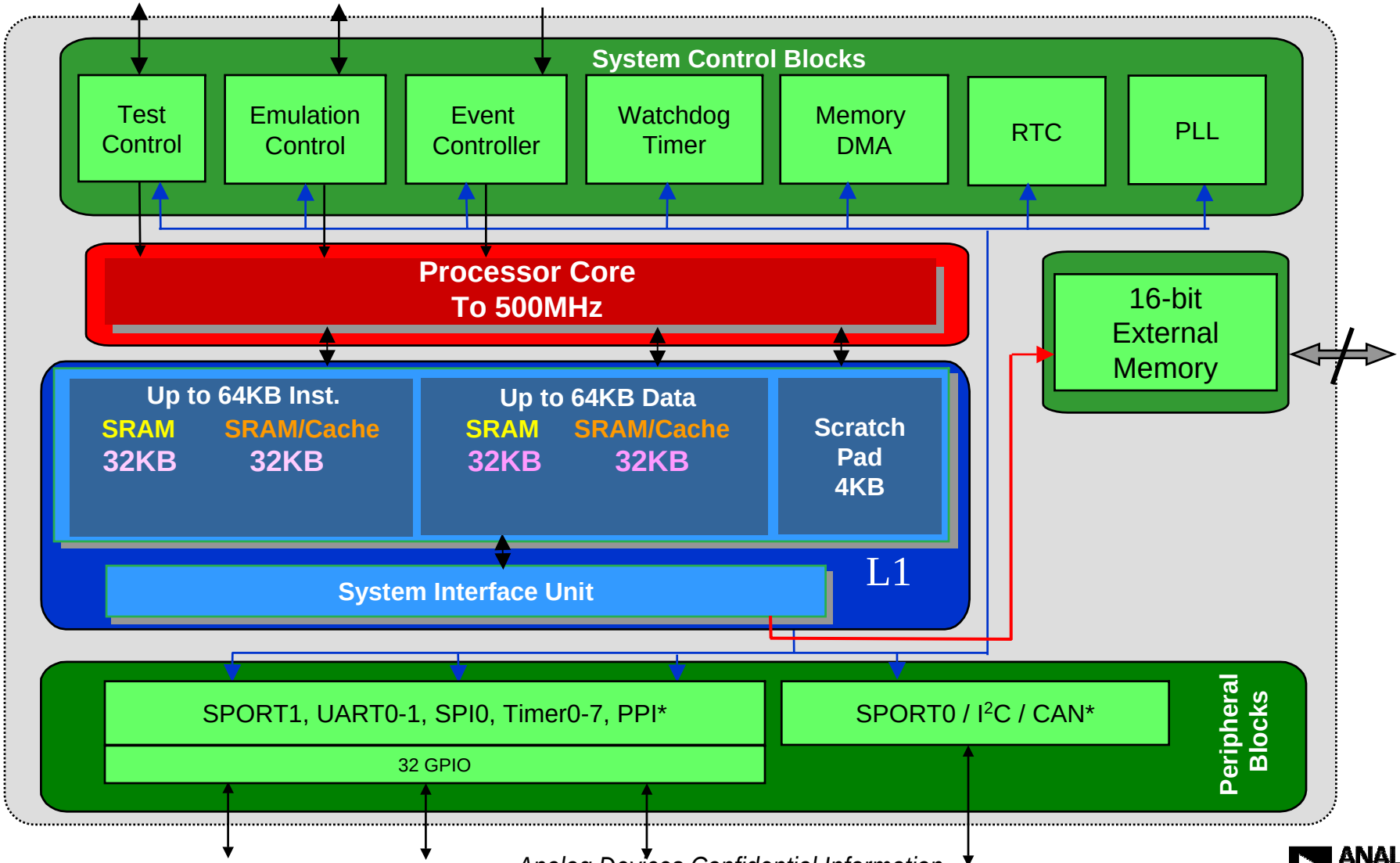


—Analog Devices Confidential Information—

ADSP-BF561 Dual-Core Blackfin – Available Now



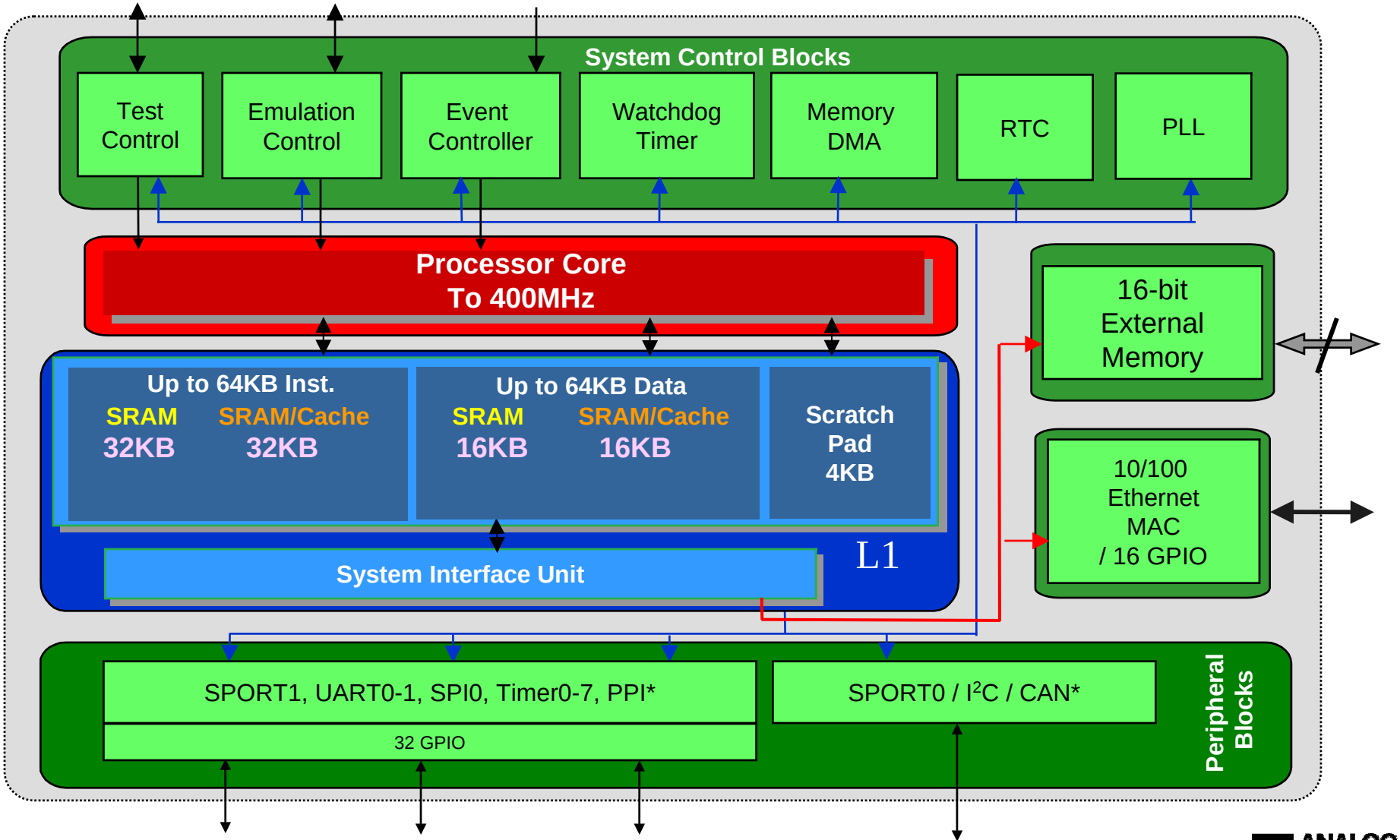
Blackfin – ADSP-BF534 – Available Now



—Analog Devices Confidential Information—

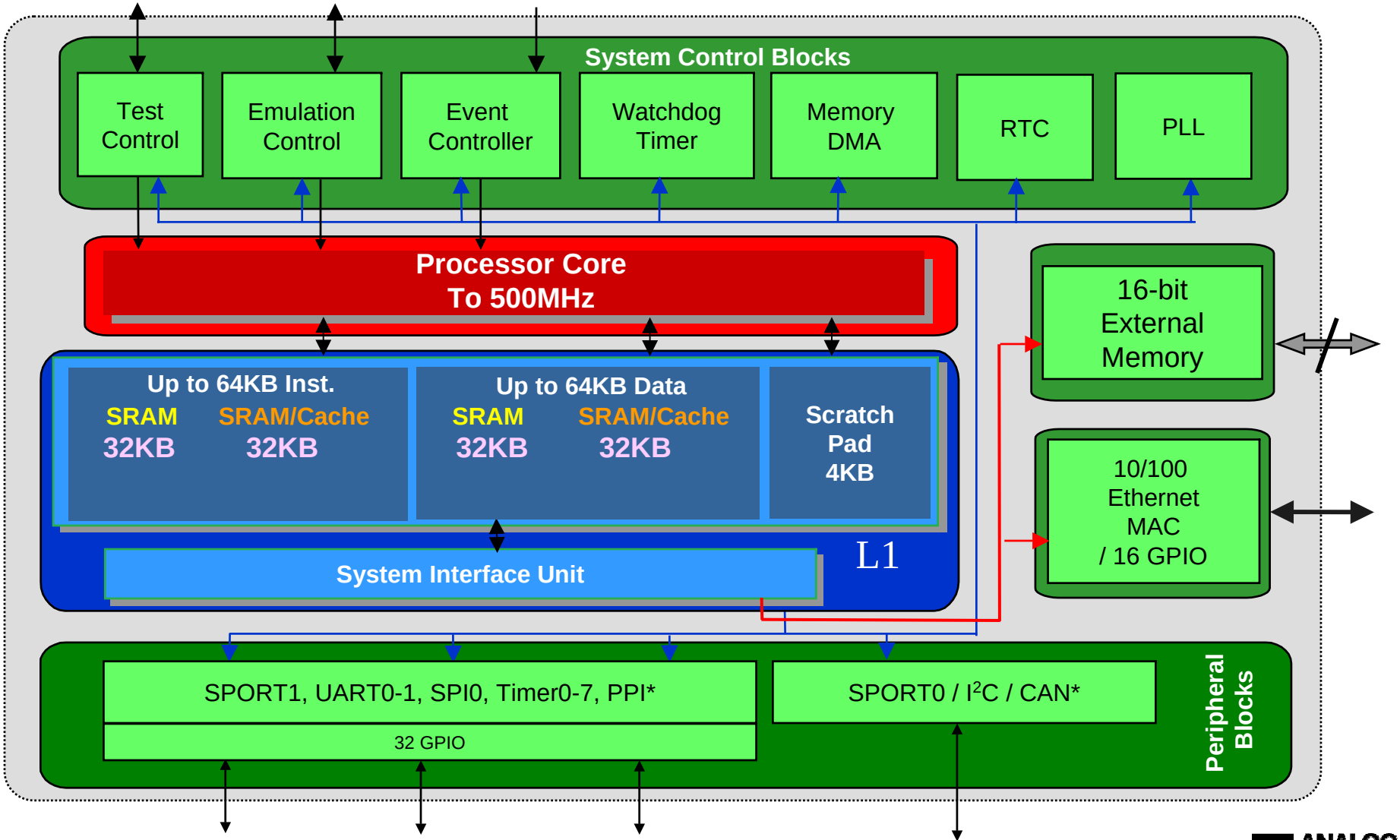
ANALOG DEVICES

Blackfin – ADSP-BF536 – Available Now



—Analog Devices Confidential Information—

Blackfin – ADSP-BF537 – Available Now



—Analog Devices Confidential Information—

Blackfin Operating System Support

- Basic Needs
- Limited Budget
- FREE with VisualDSP++™

- Media / Web centered
- Embedded XML
- #1 TCP/IP Stack in World

- OSEK Compliant
- Safety Critical

- Performance Driven
- Minimal Code Size

- De facto Std in Academic World
- Broad User Community
- Free Connotation

- Comprehensive Product Portfolio beyond Kernel
- Comprehensive CPU coverage for easy switch

- Broad Coverage and Highly Integrated

- Consumer Media
- Audio/Video
- Network Connected

- Automotive
- Telematics

- Consumer
- Media / STB
- PC & Peripheral

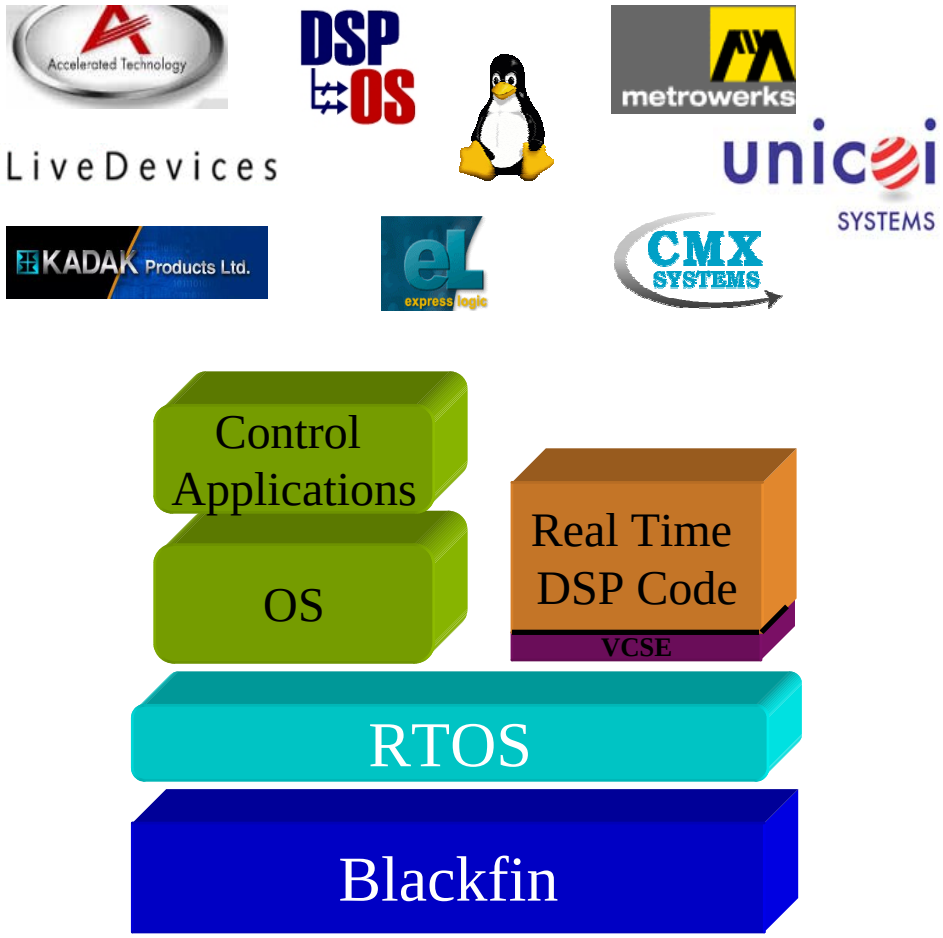
- Traditional MCU
- From Desktop to Embedded Devices

- Consumer
- Telecomm
- Industrial
- Networking

—Analog Devices Confidential Information—

ANALOG DEVICES

Operating Systems



Real Time Operating Systems

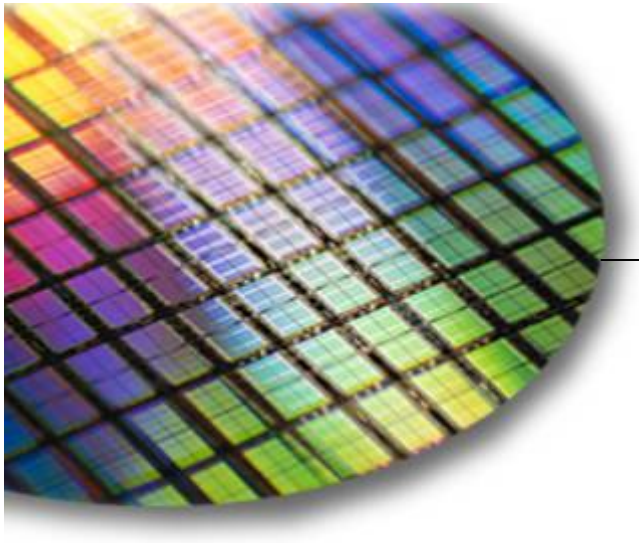
- VDK from ADI
- Unicoi Fusion RTOS
- Nucleus PLUS
- ThreadX
- CMX
- Live Devices
- uITRON (API)

Operating Systems

- Embedded Linux (BF535) BF531/2/3 – in development

Networking Stacks

- Kadak Kwik-Net
- Unicoi Fusion Net
- Net-X



The World Leader in High Performance Signal Processing Solutions

Section 2

Introduction to VisualDSP++

VisualDSP++ 4.0

- **VisualDSP++ is an integrated development environment that enables efficient management of projects.**
  - **Key Features Include:**
    - Editing
    - Building
      - Compiler, assembler, linker
    - Debugging
      - Simulation, Emulation, EZ-KIT
      - Run, Step, Halt
      - Breakpoints, Watchpoints
      - Advanced plotting and profiling capabilities
      - Pipeline and cache viewers

—Analog Devices Confidential Information—

ANALOG DEVICES

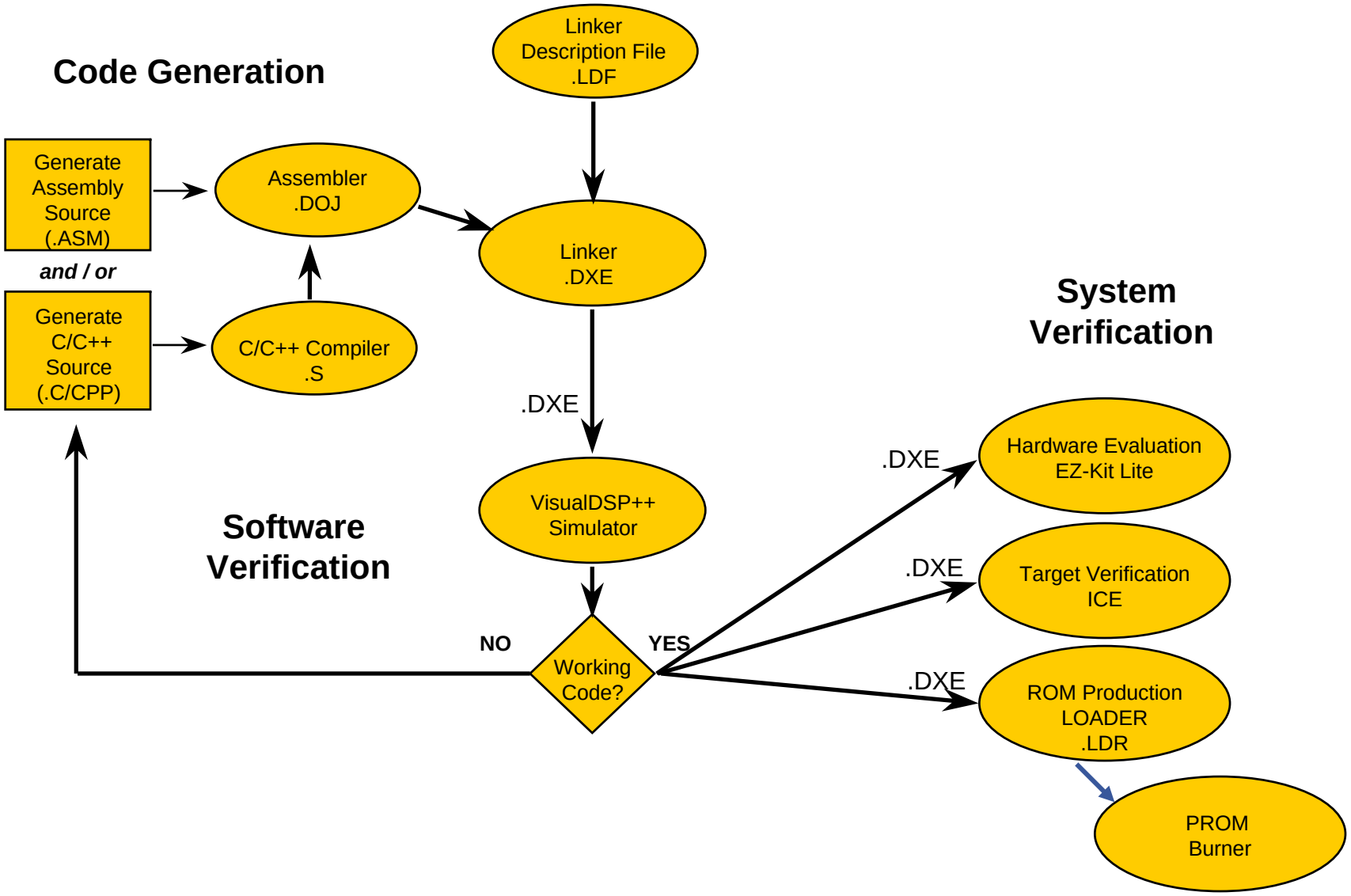
VisualDSP++

- **What comes with VisualDSP++?**
  - **Integrated Development and Debugger Environment (IDDE), C/C++ Compiler, Assembler, Linker, VDK, Emulation and Simulation Support, On-line help and documentation**
    - Part #: VDSP-BLKFN-FULL
    - Floating License Part #: VDSP-BLKFN-PCFLOAT
- **VisualDSP++ is a common development environment for all ADI processor families**
  - **Blackfin**
    - ADSP-BF5xx
  - **TigerSharc**
    - ADSP-TSxxx
  - **Sharc**
    - ADSP-21xxx
  - Each processor family requires a separate license

Features of VisualDSP++ 4.0

- **Integrated Development and Debugger Environment (IDDE)**
  - **Multiple workspaces, projects, project groups**
- **Project Wizard**
  - **Create/configure a DSP project**
- **High level language support including C and C++**
- **Expert Linker**
  - **Graphical support for managing linker description files**
  - **Code profiling support**
- **Easy to use Online Help**
- **BTC (Background Telemetry Channel) Support**
  - **Data Streaming and Logging**
- **Easy to test and verify applications with scripts (TCL, VB, Java)**
- **VisualDSP++ RTOS/Kernel/Scheduler (VDK)**
- **Integrated Source Code Control**
- **Device Drivers and System Services**

Software Development Flow



Invoking the Software Tools

- **Software tools may be configured and called by the IDDE**
  - Software tools are configured via property pages
  - The IDDE calls the software tools it needs to complete the build
    - GUI front end to a command line ‘make’ utility
- **Software tools can be invoked from a Command line**
  - C Compiler: *ccblkfn* sourcefile -switch [-switch...]
  - Assembler: e*asmblkfn* sourcefile -switch [-switch...]
  - Linker: *linker* object [object…] -switch [-switch…]
  - Loader: *elfloader* executable -switch [-switches...]
- **For the complete list of switches see the appropriate tools manual**

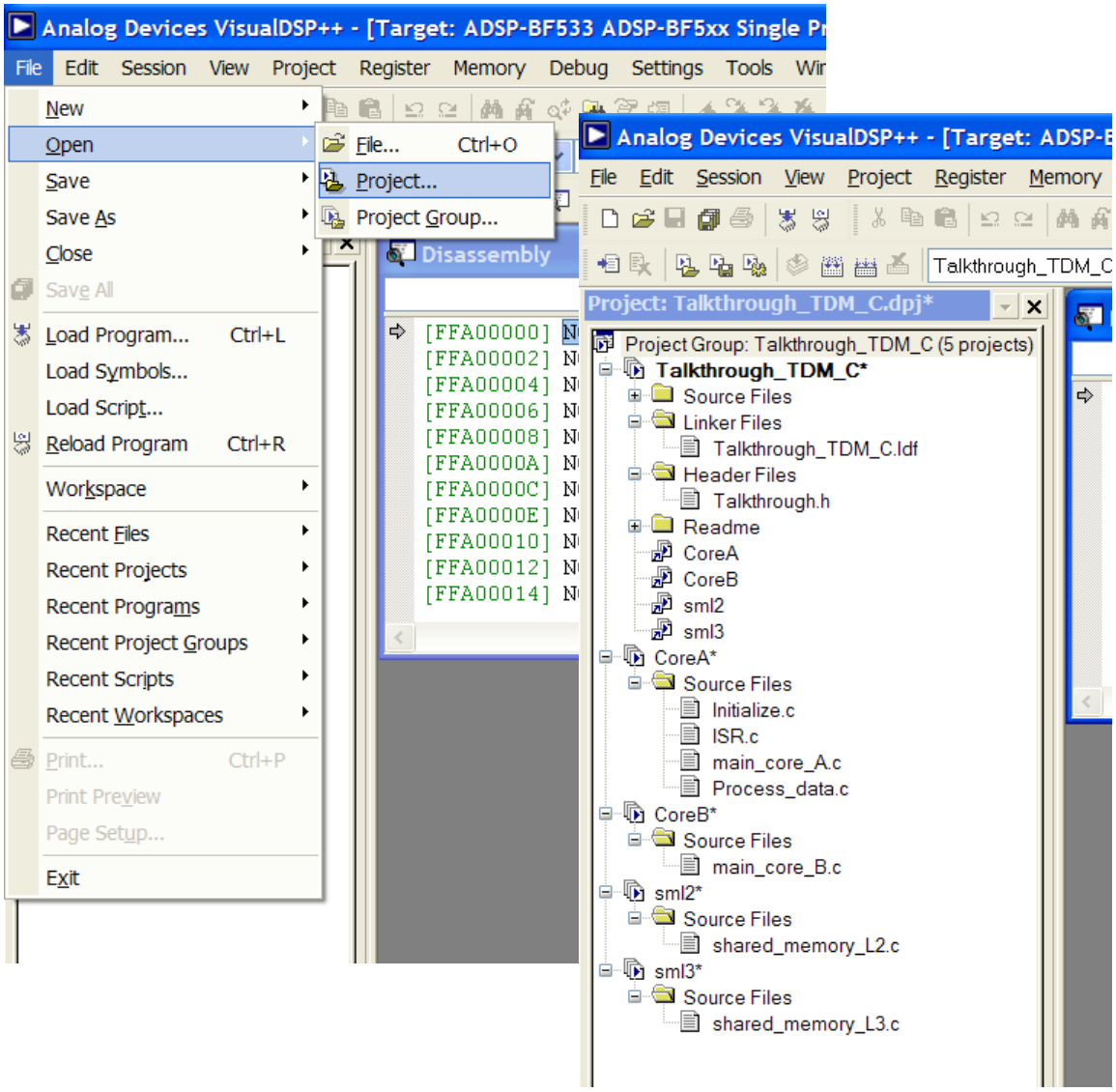
Integrated Development and Debugger Environment (IDDE) Features

- **IDDE allows one to manage the project build**
- **The user configures the project and the development tools via property pages**
- **Project Property pages configure the project**
  - Project Property Page
  - General Property Page
  - Pre Build Property Page
  - Post Build Property Page
- **Development Tools Property Pages are used to configure the development tools**
  - Assembler Property Page
  - Compiler Property Page
  - Linker Property Page
  - Loader Property Page

ANALOG DEVICES

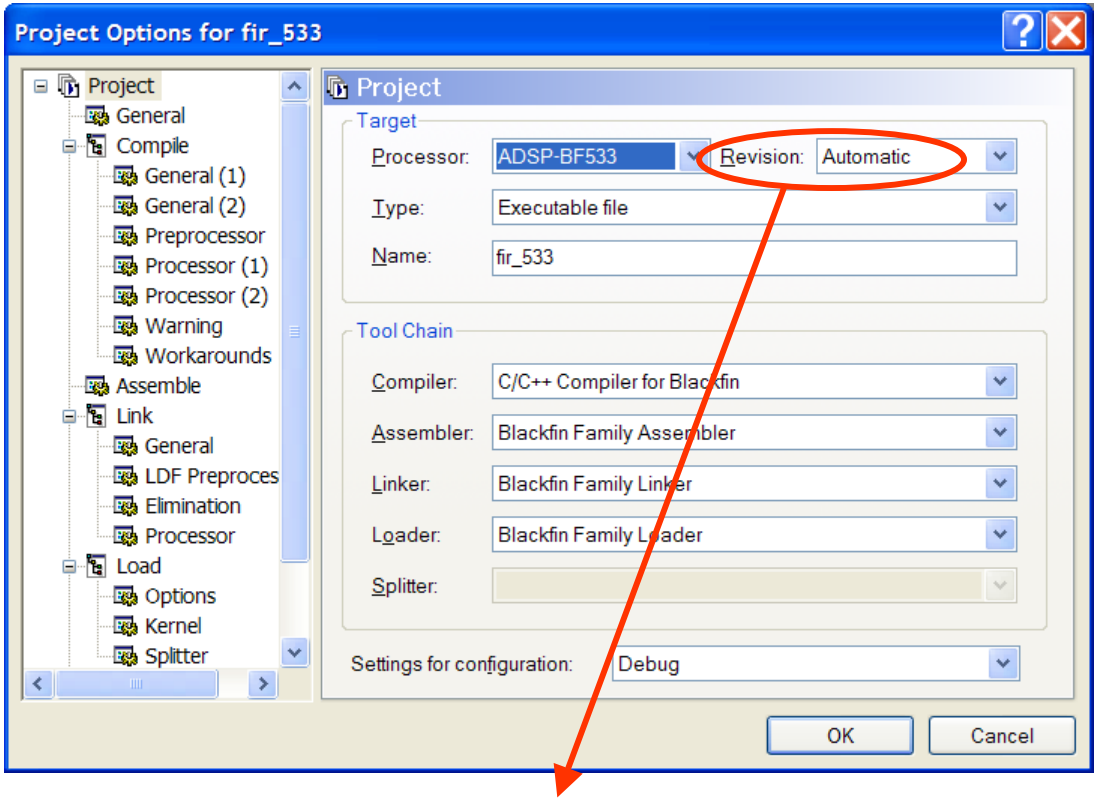
Project Development

- **Create a project**
  - All development in VisualDSP++ occurs within a project.
  - The project file (.DPJ) stores your program's build information: source files list and development tools option settings
  - A project group file (.DPG) contains a list of projects that make up an application (eg ADSP-BF561 dual core application)



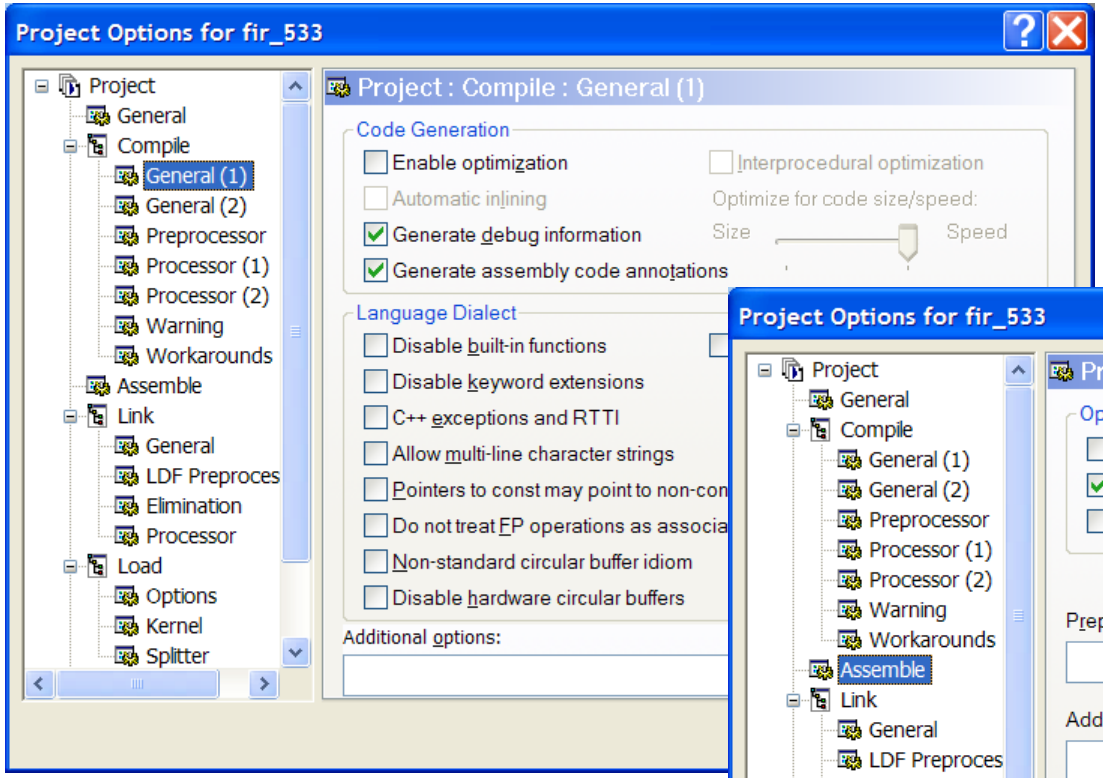
Project Property Page

- **Configure project options**
  - Define the target processor and set up your project options (or accept default settings) before adding files to the project.
  - The Project Options dialog box provides access to project options, which enable the corresponding build tools to process the project's files correctly

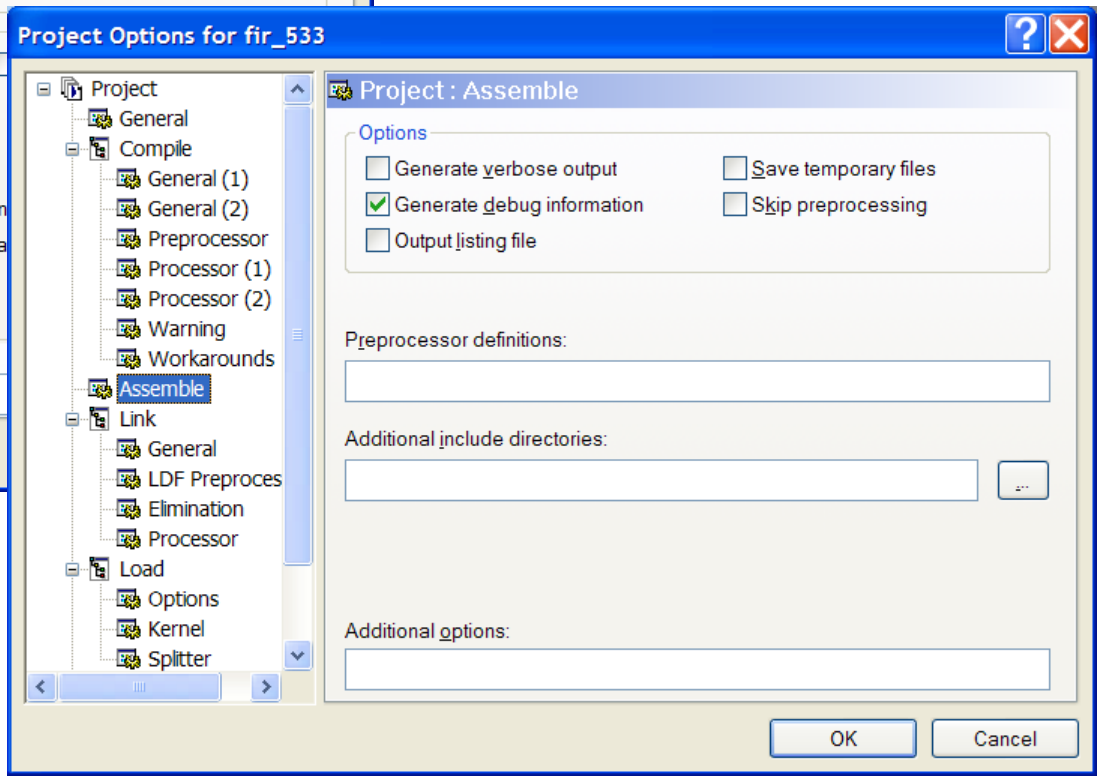


Enable building for a specific revision of silicon
- No need to specify '-si-revision' switch
- Automatic will attempt to determine revision of the attached target
- or specify a specific rev level (eg 0.3)

Property Pages

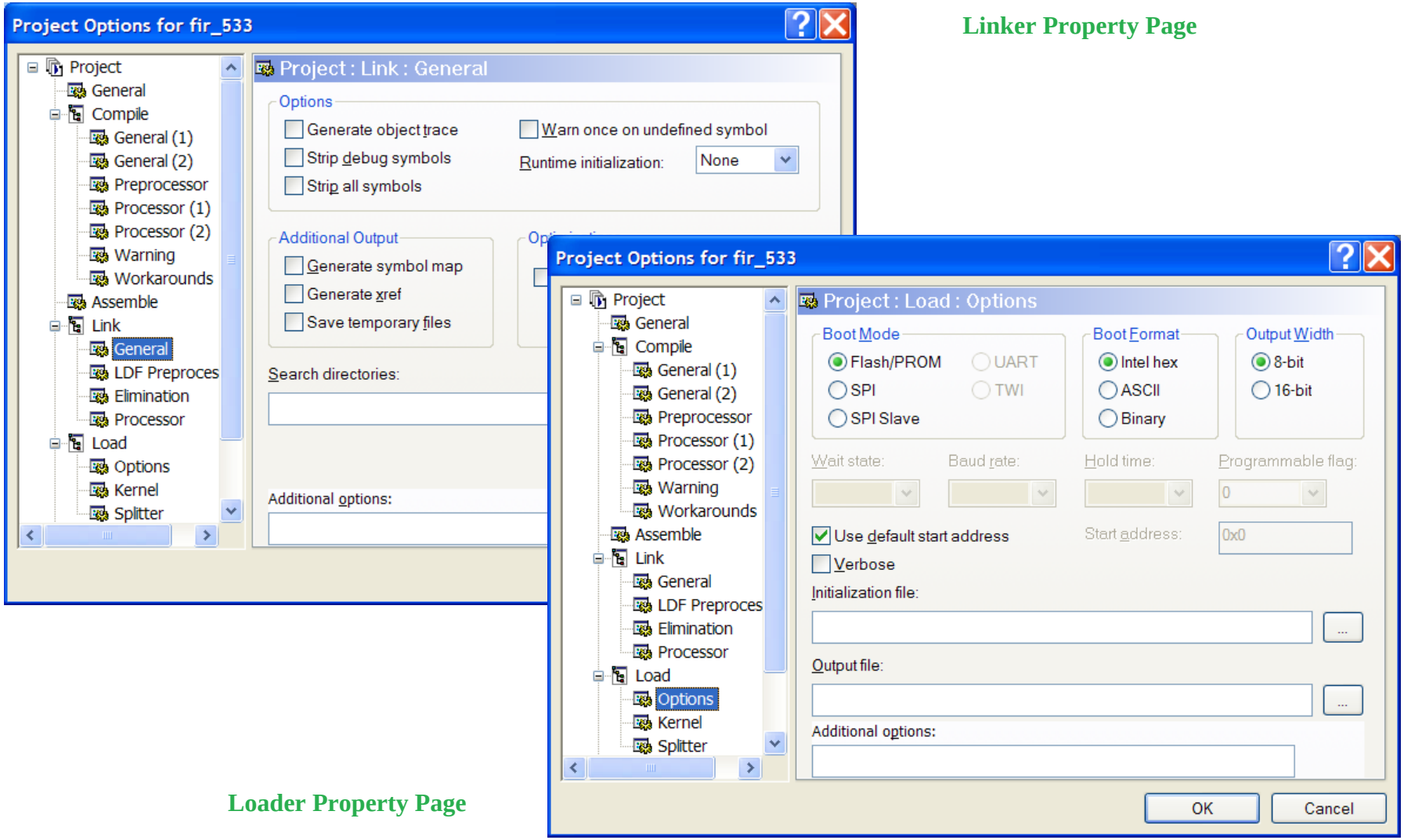


C/C++ Compiler Property Page



Assembler Property Page

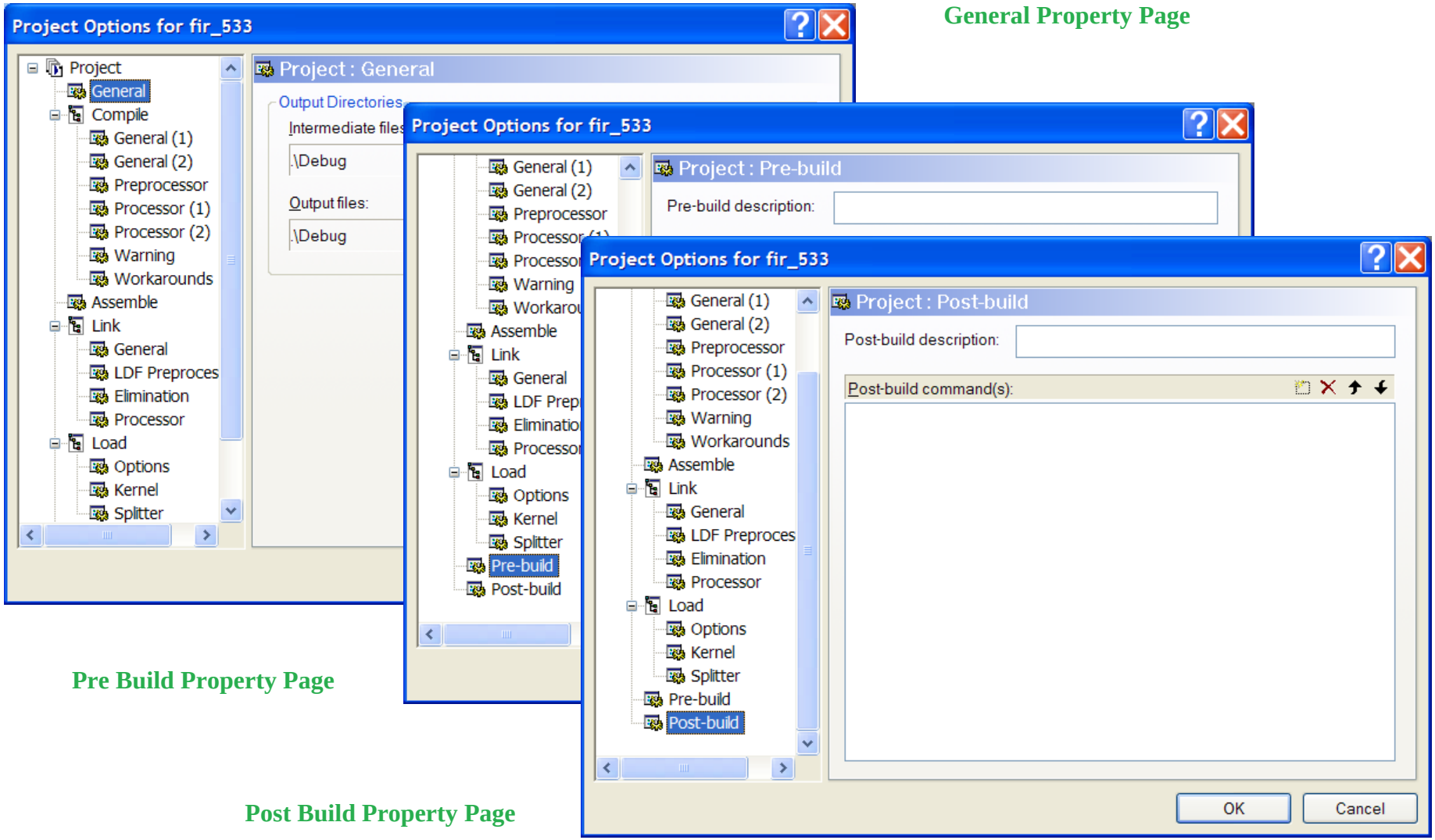
Property Pages



Linker Property Page

Loader Property Page

Property Pages



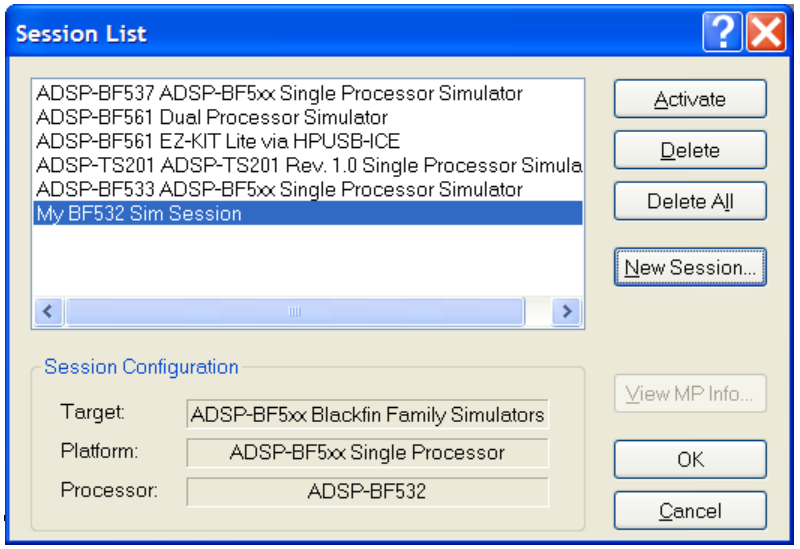
General Property Page

Pre Build Property Page

Post Build Property Page

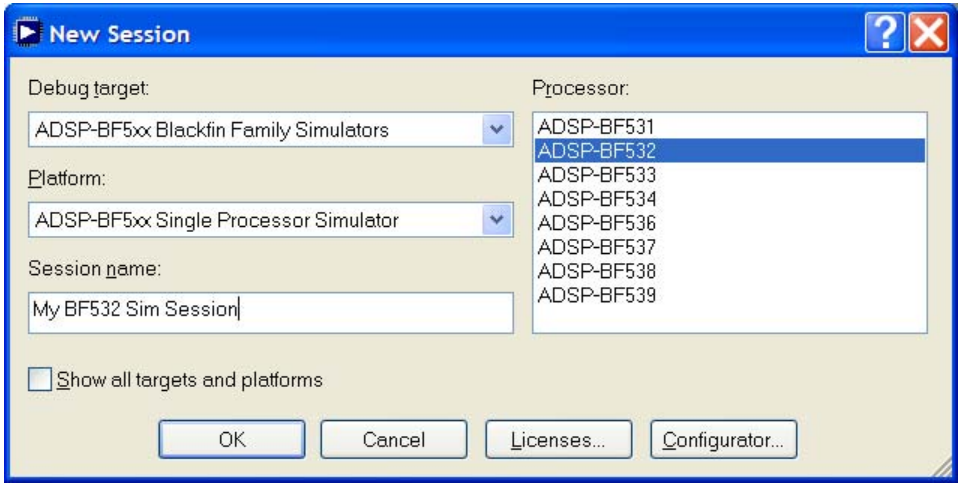
Selecting VisualDSP++ Sessions

- **Sessions define Debug Environments**
- **Select Sessions pull down menu**
  - Choose Sessions List
  - Select Session to activate
- **Define New Session from Session List**
  - Select New Session
  - Configure session as required e.g.



Debug target : ADSP-BF53x Family Simulator
Platform : ADSP-BF53x Single Processor Simulator
Session name : ADSP-BF533 ADSP-BF53x Single Processor Simulator

- **Click OK**
  - Session name will appear in Session List
- **Click Activate**
  - IDDE session will open



—Analog Devices Confidential Information—

Debug Features

- **Single Step**
- **Run**
- **Halt**
- **Set Breakpoints**
- **Register Viewing**
- **Memory**
  - **Viewing**
  - **Plotting**
  - **Dump/Fill**
- **Code Optimization Utilities**
  - **Profiling**
  - **Pipeline Viewer**
  - **Cache Viewer**
- **Compiled Simulation**
- **High Level Language debug support**
  - **Mixed mode**

Online Help

- **Fully searchable and indexed online help**
- **Includes quick overviews on using VisualDSP++ and all of its features.**
- **Excellent supplement to the manual for things that are better represented visually such as what various plot windows should look like.**
- **Customizable by using the “Favorites” window**

On Line Help Example

VisualDSP++ 4.0 Help

Hide Locate Back Forward Home Print Options

Contents Index Search

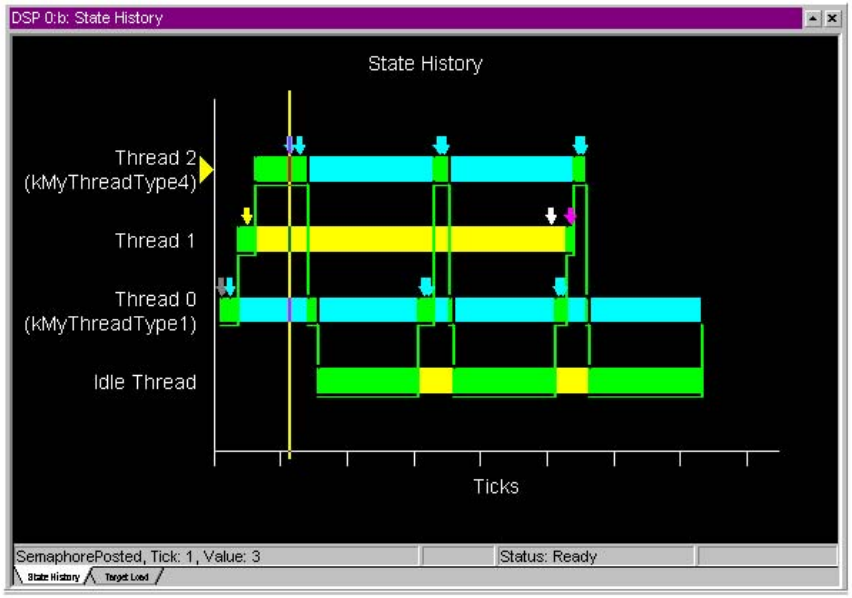
Type in the keyword to find:

VDK State History window, openin

- VDK State History window
  - opening
  - operations
  - status bar
- VDK Status window
  - about
  - adding a marshalled messag
  - adding boot threads
  - adding dependent event bits
  - adding device drivers
  - adding device flags
  - adding event bits
  - adding events
  - adding imported projects
  - adding interrupts
  - adding memory pools
  - adding round-robin priorities
  - adding routing threads to pro
  - adding semaphores
  - adding thread types
  - changing boot thread order
  - configuration
  - data cursor
  - deleting boot I/O objects
  - deleting boot threads
  - deleting device drivers
  - deleting device flags
  - deleting event bits
  - deleting events
  - deleting interrupts
  - deleting memory pools
  - deleting round-robin priorities
  - deleting semaphores

Display

VDK State History Window Operations



Status Bar

The status bar (bottom of plot) of the State History page of the VDK State History window shows the event's details and thread status when the data cursor is enabled. Event details include the event type, the tick when the event occurred, and an event value.

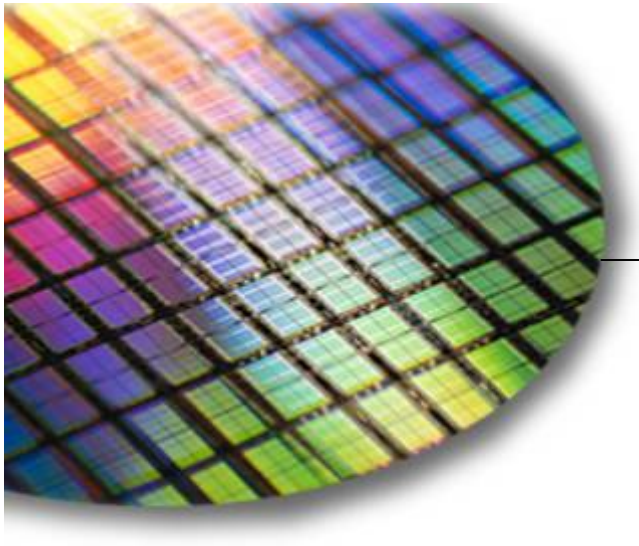
The value for a thread-switched event indicates the thread being switched in or out.

The status bar indicates thread status for the active location.

Data Cursor

What is VDK ?

- **VDK is a kernel not an operating system**
- **VDK comprises:**
  - **VDK libraries**
  - **VDK specific ldf files**
  - **Include files**
  - **Template files**
- **Overheads**
  - **Memory overhead**
  - **Minimum memory requirement is platform dependent**
  - **Footprint is one of the most important metrics for a RT kernel**
  - **MIPS overhead**



The World Leader in High Performance Signal Processing Solutions

Coding Guidelines for Achieving Optimal C Performance on Blackfin

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Strategic Objective:
Make C as fast as assembler!

Advantages:

C is much cheaper to develop.

C is much cheaper to maintain.

C is comparatively portable.

- **Disadvantages:**

ANSI C is not designed for DSP.

DSP processor designs usually expect assembly in key areas.

DSP applications continue to evolve.

ANALOG DEVICES

Pillars of Effective Programming

- **Understand Underlying Hardware Capabilities**
- **Discover What Compiler Can Provide**
- **Design Program Effectively**
  - **general choice of algorithm**
  - **choice of data representation**
  - **finer low-level programming decisions**

- **Usually the process of performance tuning is a specialisation of the program for particular hardware. It may grow larger or more complex and is less portable.**

—Analog Devices Confidential Information—

ANALOG DEVICES

Analog C Compiler (VDSP++ 4.0)

- **State-of-the-art optimizer.**
  - **Provides flexibility**
  - **Ease of adding architecture-specific optimizations**
- **Exploitation of explicit parallelism in the architecture**
  - **Vectorization – exploiting wide load capabilities**
  - **Recognizing SIMD opportunities**
  - **Software pipelining**
- **Whole Program Analysis**
  - **A wider view enables the optimizer to be more aggressive.**

Optimizer improvements in VDSP++ 4.0

- **Intelligent Vectorization**
  - **More flexible, heuristic based vectorization.**
- **Unroll and Jam**
  - **Unroll outer loop and combine resulting copies of inner loop.**
- **Minimising Call Overhead**
  - **Can supply list of registers altered by a function.**

—Analog Devices Confidential Information—

Other new features with VDSP 3.5

- **long long support - 64-bit integer support**
- **Enhanced GNU compatibility features.**
- **compiler built-ins added for Blackfin video operations.**
- **ADSP-BF561 support**
- **multiple-heap support**
- **improved cache support**
- **C++ Exception Handling**
- **Profile-Guided Optimization**

Understanding Underlying Hardware

- **Isn't C supposed to be portable & machine independent?**
  - **yes, but at a price!**
  - **Uniform computational model, BUT….**
    - missing operations provided by software emulation (slow)
    - for example: C provides floating point arithmetic everywhere
  - **C is more machine-dependent than you might think**
    - for example: is a "short" 16 or 32 bits? (more later)
- **Machine's Characteristics will determine your success.**

C programs can be ported with little difficulty.

But if you want high efficiency, you can't ignore the underlying hardware

—Analog Devices Confidential Information—

Evaluate Algorithm against Hardware.

- **What's the native arithmetic support?**
  - **Can we use floating point hardware?**
  - **how wide is the integer arithmetic?**
    - doing 64-bit arithmetic on a 32-bit unit is slow
    - doing 16-bit arithmetic on a 32 bit part is awkward
  - **Can we use packed data operations?**
    - 2x16 arithmetic might be ideal for your application (more computation per cycle, less memory usage)
    - implications for data types, memory layout, algorithms
- **What is the computational bandwidth and throughput?**
  - **what are the key operations required by your algorithm?**
  - **(macs?, loads?, stores?….)**
  - **how fast can the computer perform them?**

—Analog Devices Confidential Information—

ANALOG DEVICES

DSP's Present Some Unique Problems

Special Aspects of Digital Signal Processors:

- **Reduced memory**
- **Extended precision accumulators**
- **Specialized architectural features**
  If not well modeled by C : lose portability and efficiency
  - Example: Zero overhead loop – good
  - Fractional arithmetic - problem.
- **mathematical focus (historically not C's orientation)**

Features which compiler must exploit

- **Efficient Load / Store Operations in Parallel**
- **Utilize multiple Data-paths; SISD, SIMD, MIMD operations**
- **minimize memory utilization**

ANALOG DEVICES

C and the Compiler

- **C provides common computational model**
  - **portability**
  - **higher level**
- **Compiler's job: map this to a particular machine**
  - **tries for optimal use of instructions**
  - **supplement by instruction sequences or library calls**
- **Optimizer improves performance**
  - **do things less often, more cheaply**
  - **try to utilize resources fully**
- **Optimizing Compiler has Limited Scope**
  - **will not make global changes**
  - **will not substitute a different algorithm**
  - **will not significantly rearrange data or use different types**
  - **correctness as defined in the language is the priority**

—Analog Devices Confidential Information—

ANALOG DEVICES

Overview of Compilation

Compiler:

(1) makes a straightforward translation

- fully sequential
- each individual step as written

(2) then improves it (optimization)

- **transforms it into an *equivalent* one**
  - hopefully faster and smaller
  - must get same “answers”
- **Simple Guiding Principle:**
  - Avoid Work
  - Reduce Generality
  - Do things in parallel

This form provides clearest debugging

—Analog Devices Confidential Information—

ANALOG DEVICES

Summary:
How to go about increasing performance.

1. **Work at high level first**
   most effective -- maintains portability
   - improve algorithm
   - make sure it's suited to hardware architecture
   - check on generality and aliasing problems
2. **Look at machine capabilities**
   - may have specialized instructions (library/portable)
   - check handling of DSP-specific demands
3. **Non-portable changes last**
   - in C?
   - in assembly language?
   - always make sure simple C models exist for verification.

- **Compiler will improve with each release**

ANALOG DEVICES

Choose!
Optimized C or Out of the Box C?

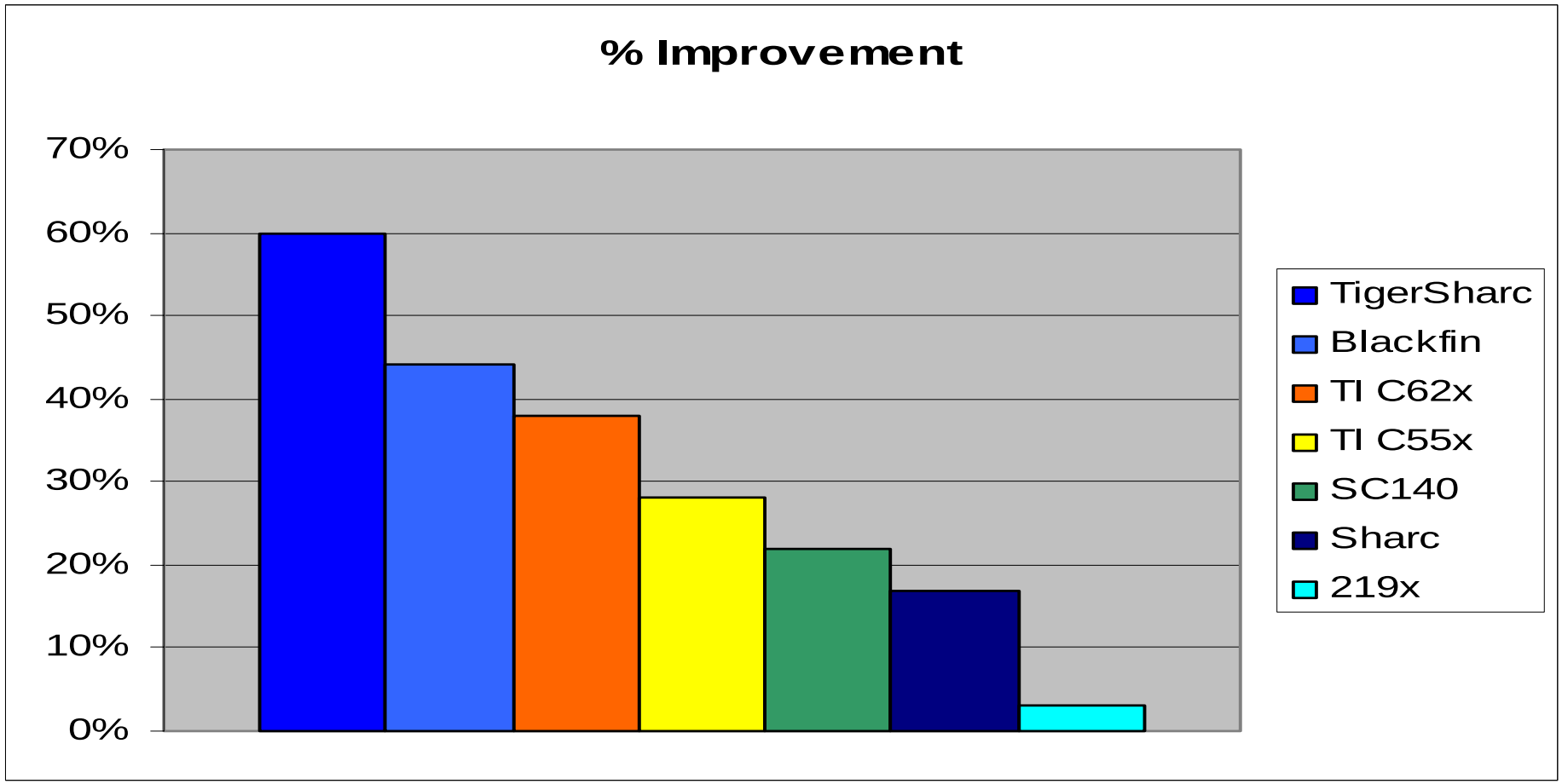
- **OTB or "out of the box" C is portable code.**
  But most platforms allow some "elaboration" of the source.

- **#pragmas. - (Compiler specific assertions.)**
- **__builtin functions.**
- **Memory qualifiers – const, restrict, volatile, bank.**

- **These can specify alignment, cycle iteration count, SIMD, memory type. Or access specific machine instructions one to one.**
- **Optimized C can go very much faster than "out of the box C".**

ANALOG DEVICES

OTB C compilers are improving rapidly. EDN: Improvement in the last 2 years.



Use the Optimizer!

- **There is a massive effect from optimization on a DSP platform. (Much more than on RISC chips)**
- **Non – optimised code is up to 20 times slower.**
- **Sliding scale from control code to DSP inner loop.**
- **Non-optimized code is only for debugging the algorithm.**
  - **(You can also perform limited debugging optimized, with –O –g, which gives access to global variables, function names and line numbers.)**

—Analog Devices Confidential Information—

Un-Optimized Code for Blackfin

Unoptimised assembly:

```
  [FP+ -8] = R7; ._P1L1:
  R3=[FP+ -8];
  R2 =  150 (X);
  CC = R3 < R2;
  IF !CC JUMP  ._P1L3 ;
  R3 <<=  1;
  P2 = R3 ;
  P0=[FP+ 8];
  P0 = P0 + P2;
  R1=W[P0+ 0] (X);
  R0=[FP+ -8];
  R0 <<=  1;
  P1 = R0 ;
  P2=[FP+ 12];
  P2 = P2 + P1;
  R7=W[P2+ 0] (X);
  R7 *= R1 ;
  R1=[FP+ -4];
  R0 = R1 + R7;
 [FP+ -4] = R0;
  R3=[FP+ -8];
  R3 <<=  1;
  P0 = R3 ;
  P1=[FP+ 12];
  P1 = P1 + P0;
  R1=W[P1+ 0] (X);
  R7=[FP+ -8];
  R7 <<=  1;
  P2 = R7 ;
  P1=[FP+ 12];
  P1 = P1 + P2;
  R3=W[P1+ 0] (X);
  R3 *= R1 ;
  R1=[FP+ 16];
  R7 = R1 + R3;
  [FP+ 16] = R7;
  R3=[FP+ -8];
  R3 +=  1;
  [FP+ -8] = R3;
  JUMP ._P1L1;
```

Loop control increment, test & exit

Load A[I]

Load B[I]

Sum += A[I]* B[I]

Load B[I]

Load B[I]

B[I] * B[I]

Increment I

Repeat Loop

The source code:

```
for (i = 0; i < 150; i++) {
   dotp += b[i] * a[i];
   sqr  += b[i] * b[i];
 }
```

The Optimised assembly

- easier to understand!

```
LSETUP (._P1L2 , ._P1L3-8) LC0=P1;

._P1L2:
  A1+= R0.H*R0.H, A0+= R0.L*R0.H (IS)
          || R0.L = W[I1++]
          || R0.H = W[I0++];
._P1L3:
```


ANALOG DEVICES

General Principles of Optimizer

The Optimizer Looks at Each Operation:

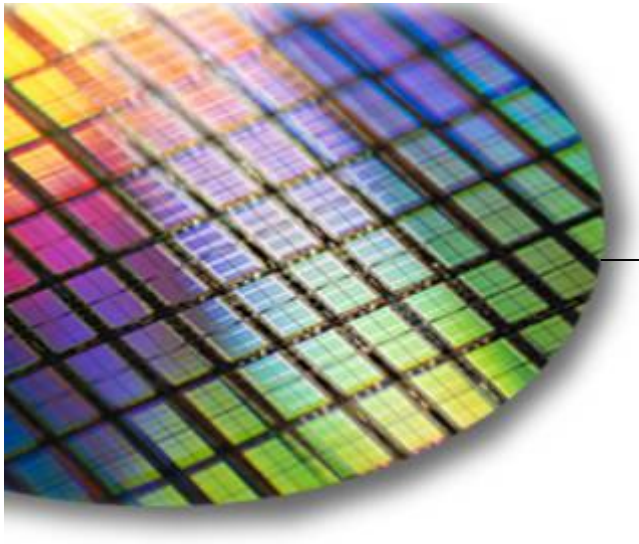
- **Try not to do it at all**
  - perhaps not actually needed
  - calculate at compile-time
  - re-use previous calculation
- **Do it more cheaply**
  - avoiding storing in memory
- **Do it more efficiently**
  - use special resources
  - do more than one thing at a time
- **Loops get special attention**
  - ***Biggest Savings of All***

The compiler is your partner

You can count on certain optimizations being done

Compiler command line options

-O	Optimize
-Oa	Optimize with auto-inlining
-Os	Optimize space sensitively
-Ov	Optimize with user control of balance between size and speed
-ipa	Whole program analysis
-save-temps	Preserves compiler output (.s)



The World Leader in High Performance Signal Processing Solutions

Leave the low level concerns to the compiler.

Leave basic operations to the compiler.

```
(1)   a = b * c;
(2)   d = a + f;
(3)   a = b - g;
```

Value of 'a' can be used directly from register; eliminate load from memory

New value assigned to 'a', so value stored at (1) is not used; eliminate the store to memory

Straightforward code

```
R2 = [b];
R3 = [c];
R1 = R2 * R3;
[a] = R1;

R1 = [a];
R6 = [f];
R4 = R1 + R6;
[d] = R4;

R2 = [b];
R7 = [g];
R1 = R2 - R7;
[a] = R1;
```

12 cycles

(already in R2)

Optimized code

```
R2 = [b];
R3 = [c];
R1 = R2 * R3;


R6 = [f];
R4 = R1 + R6;
[d] = R4;

R7 = [g];
R1 = R2 - R7;
[a] = R1;
```

9 cycles

Leave scheduling to the compiler.

```
(1)     a = b * c;
(2)     d = a + f;
(3)     a = b - g;
```

Take advantage of hardware parallelism: consider dispatching multiple instructions in one cycle

Optimized code

```
R2 = [b];
R3 = [c];
R1 = R2 * R3;

R6 = [f];
R4 = R1 + R6;
[d] = R4;

R7 = [g];
R1 = R2 - R7;
[a] = R1;
```

9 cycles

Scheduled code

```
R2 = [b];
R3 = [c];
R1 = R2 * R3,   R6 = [f];

R4 = R1 + R6,   R7= [g];

R1 = R2 - R7,   [d] = R4;

[a] = R1;
```

6 cycles

ANALOG DEVICES

Compilers understand Loops

Simple counted loop:
Use zero-overhead loop mechanism

C and D don't change during loop:
Load them into registers outside

```
for (j=0; j<N; j++) {
   P[j] =  A[j] * C  +  B[N-j-1] * D ;
   Q[j] =  A[j] * C  -  B[N-j-1] * D ;
   }
```

Combine reference with incrementing pointer

(Use post-modify addressing)

COMPILER DOES THE LOW-LEVEL WORK

—Analog Devices Confidential Information—

Addressing Operations are Fully Efficient

```
pA = &A[0];
pB = &B[0];
pP = &P[N-1];
pQ = &Q[N-1];
```

> zero-overhead loop
>
> C, D loop invariant, loaded once outside loop

```
for (j=0; j<N; j++) {
    P[j] = A[j] * C + B[N-j-1] * D;
    Q[j] = A[j] * C - B[N-j-1] * D;
}
```

*pP++

*pQ++

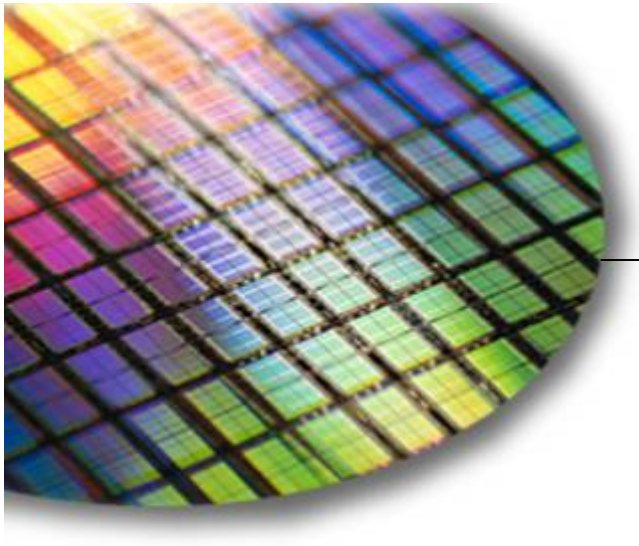
*pA++

*pB--

(loaded once, reused)

You Can Count on the Optimizer to Do This Transformation

ANALOG DEVICES



The World Leader in High Performance Signal Processing Solutions

How can we improve on the compilers effort?

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Getting Started 80:20

Find out where program spends its time.

- **80 – 20 rule**
- Measure: **Intuition is notoriously bad here: instrument, use profiler and cycle accurate simulator.**
- Loops: **Are always a good place to look.**

  Even a trivial operation can have a significant cost, if it is done often enough.

ANALOG DEVICES

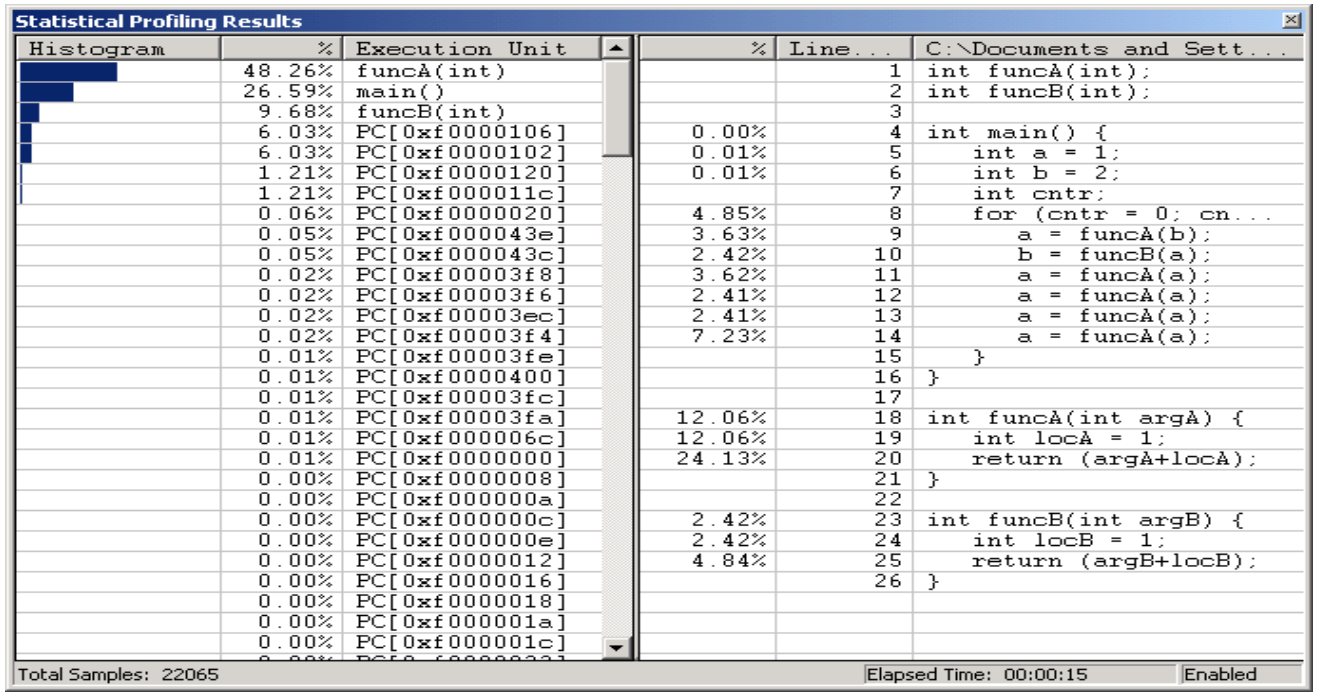
Use the Statistical Profiler

- **Statistical profiling samples the program counter of the running application and builds up a picture of where it spends its time**
- **Completely non-intrusive – no tracing code is added.**
- **Completely accurate – shows all effects, including stalls.**

- **Don’t assume you know where an application spends its time – profile it.**

VDSP Statistical Profiler

- **The profiler is very useful in C/C++ mode because it makes it easy to benchmark a system module-by-module (I.e. C/C++ function).**
- **Assembly or optimised code appears as individual instructions.**



Statistical Profiling Results

Histogram	%	Execution Unit	%	Line...	C:\Documents and Sett...
	48.26%	funcA(int)		1	int funcA(int);
	26.59%	main()		2	int funcB(int);
	9.68%	funcB(int)		3	
	6.03%	PC[0xf0000106]	0.00%	4	int main() {
	6.03%	PC[0xf0000102]	0.01%	5	int a = 1;
	1.21%	PC[0xf0000120]	0.01%	6	int b = 2;
	1.21%	PC[0xf000011c]		7	int cntr;
	0.06%	PC[0xf0000020]	4.85%	8	for (cntr = 0; cn...
	0.05%	PC[0xf000043e]	3.63%	9	a = funcA(b);
	0.05%	PC[0xf000043c]	2.42%	10	b = funcB(a);
	0.02%	PC[0xf00003f8]	3.62%	11	a = funcA(a);
	0.02%	PC[0xf00003f6]	2.41%	12	a = funcA(a);
	0.02%	PC[0xf00003ec]	2.41%	13	a = funcA(a);
	0.02%	PC[0xf00003f4]	7.23%	14	a = funcA(a);
	0.01%	PC[0xf00003fe]		15	}
	0.01%	PC[0xf0000400]		16	}
	0.01%	PC[0xf00003fc]		17	
	0.01%	PC[0xf00003fa]	12.06%	18	int funcA(int argA) {
	0.01%	PC[0xf000006c]	12.06%	19	int locA = 1;
	0.01%	PC[0xf0000000]	24.13%	20	return (argA+locA);
	0.00%	PC[0xf0000008]		21	}
	0.00%	PC[0xf000000a]		22	
	0.00%	PC[0xf000000c]	2.42%	23	int funcB(int argB) {
	0.00%	PC[0xf000000e]	2.42%	24	int locB = 1;
	0.00%	PC[0xf0000012]	4.84%	25	return (argB+locB);
	0.00%	PC[0xf0000016]		26	}
	0.00%	PC[0xf0000018]			
	0.00%	PC[0xf000001a]			
	0.00%	PC[0xf000001c]			

Total Samples: 22065 | Elapsed Time: 00:00:15 | Enabled

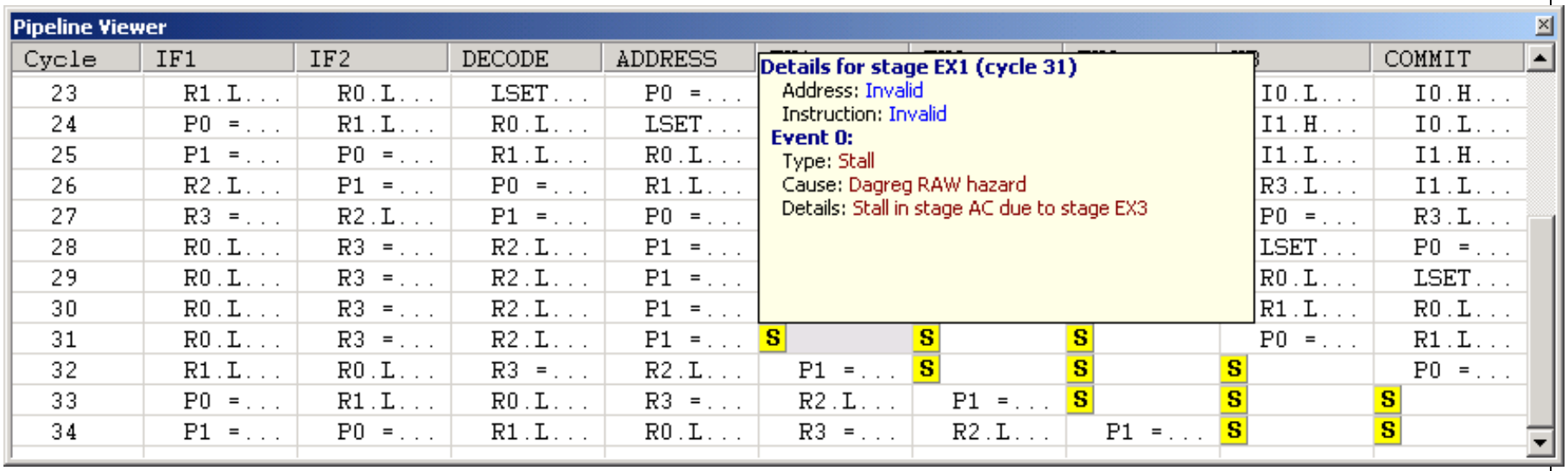
- **Linear Profiler is also available for the simulator.**

Look closely at cycles in critical areas.

- **Cycle Accurate Simulator.**
  - **Step through the code identified by the Statistical profiler. Watch the Cycle counter.**
- **Pipeline Viewer.**
  - **Close in on causes of stalls with the pipeline viewer.**

VDSP Pipeline Viewer

- **Accessed through View->Debug Windows->Pipeline Viewer in a simulator session (not available in emulator)**



How about the “pipeline”?

- **Deep pipeline processors:**
  - pipelines do badly on conditionally branching code also on table lookup
  - sometimes branches can be avoided by using other techniques
- **Is there a latency associated with computations? (results not ready on next cycle)**
  - latency can be hidden within a loop
  - hiding latencies involves loop setup overhead -- a problem if iteration counts are low
- **C Compiler will do its best, but inherent hardware limitations will always influence the outcome**
- **Pipeline is FULLY interlocked and interruptable!**

ANALOG DEVICES

Blackfin Pipeline Latencies

1. **Multiply/Video Operation Latencies (One stall)**

```
R0 = R4;
STALL
R2.H = R1.L * R0.H;
```

2. **Load to DAG Latencies (Three stalls)**

```
P3 = [SP++];
STALL
STALL
STALL
R0 = P3;
```

3. **Sub-bank access collision (One stall)**

```
STALL
R1 = R4.L * R5.H (IS) || R3 = [I0++] || R4 = [I1++];
```

Blackfin Pipeline Latencies (2)

- **4. Instruction flow dependencies**
  Correctly predicted branch (4 stalls)
  Incorrectly predicted branch (8 stalls)

- **5. Store buffer load collision**

```
W[P0] = R0;
STALL
R1 = W[P0];
```

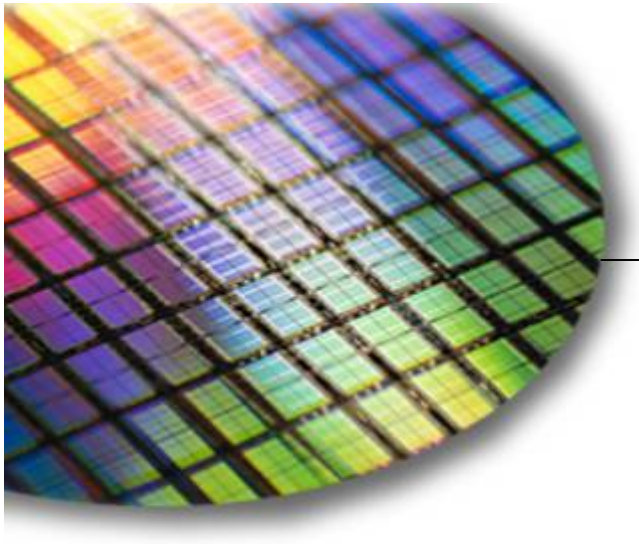
- **6. Hardware loop latencies**
  - **(example is instructions between lsetup and loop top.**

```
LSETUP(top, bottom) LC0 = P0;
  (3 STALLS)
  P0 = R0;
top:
```


ANALOG DEVICES

Latency -> affects programming style

- **Take care with structure depth.**
  - p->q->z is inefficient to access.
  - (And hard on pointer analysis. What data does this reference?)
- **Take care with Table Lookup.**



The World Leader in High Performance Signal Processing Solutions

Data types

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Native C Data Types on Blackfin

char	**8-bit signed**
unsigned char	**8-bit unsigned**
short	**16-bit signed integer**
unsigned short	**16-bit unsigned integer**
int	**32-bit signed integer**
unsigned int	**32-bit unsigned integer**
long	**32-bit signed integer**
unsigned long	**32-bit unsigned integer**

- **float (32-bit), double (32-bit), long long (64-bit) and unsigned long long (64-bit) are not supported by the hardware.**

An efficient floating Point Emulation.

Measurement in cycles		
	TI 55xx	BF532
Multiply	330	95
Add	163	108
Subtract	195	145
Divide	655	246
Sine	5341	2164
Cos	5942	2029
Square Root	5836	316
Smaller is better.		

And then add in MHZ advantage.

Note: Our Square root uses a better algorithm!

Smaller is better!

ANALOG DEVICES

Wide support for Fractional processing.

- **The Blackfin instruction set includes a number of operations which support fractional (or fract) data. The instructions include:**
  - **saturating MAC/ALU/SHIFT instructions**
  - **MAC shift correction for fractional inputs**
- **The compiler and libraries provide support for fractional types:**
  - **fractional builtins**
  - **fract types fract16 and fract32**
  - **ETSI**
  - **C++ fract class**
- **Fractional arithmetic is a hundred times faster than floating!**

ANALOG DEVICES

ETSI Builtins – fully optimised Fractional arithmetic to a standard specification.

- **European Telecommunications Standards Institute's fract functions carefully mapped onto the compiler built-ins.**

add()	**sub()**	**abs_s()**	**shl()**
shr()	**mult()**	**mult_r()**	**negate()**
round()	**L_add()**	**L_sub()**	**L_abs()**
L_negate()	**L_shl()**	**L_shr()**	**L_mult()**
L_mac()	**L_msu()**	**saturate()**	**extract_h()**
extract_l()	**L_deposit_l()**	**L_deposit_h()**	**div_s()**
norm_s()	**norm_l()**	**L_Extract()**	**L_Comp()**
Mpy_32()	**Mpy_32_16()**		

- **Immediate optimisation of ETSI standard codecs.**
- **Highly recommended!**

Pointers or Arrays?

- **Arrays are easier to analyse.**

```
void va_ind(int a[], int b[], int out[], int n) {
        int i;
        for (i = 0; i < n; ++i)
                out[i] = a[i] + b[i];
}
```

- **Pointers are closer to the hardware.**

```
void va_ptr(int a[], int b[], int out[], int n) {
        int i,
        for (i = 0; i < n; ++i)
                *out++ = *a++ + *b++
}
```

- **Which produces the fastest code?**

ANALOG DEVICES

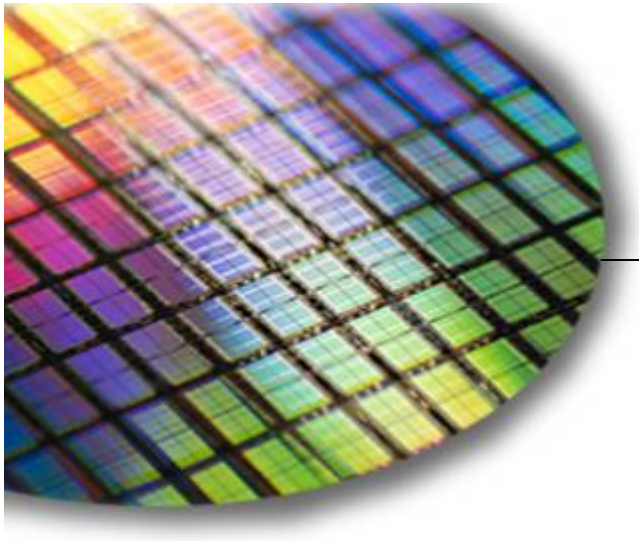
Pointers or Arrays? 2

- **Often no difference.**
- **Sometimes one version may do better for an algorithm.**
- **Not always the same style that wins.**

- **Start using array notation as easier to understand.**
- **Array format can be better for alias analysis in helping to ensure no overlap.**
- **If performance is unsatisfactory try using pointers.**
- **Outside critical loops stay with array notation.**

—Analog Devices Confidential Information—

ANALOG DEVICES



The World Leader in High Performance Signal Processing Solutions

Tricks
(useful transformations)

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Avoid Division.

- **There are no divide instructions – just supporting instructions.**
- **Floating or integer division very costly**
- **Remember Modulus(%) also implies division.**

Get Division out of loops wherever possible.

ANALOG DEVICES

Exception – Division by powers of 2.

- **Division by power of 2 rendered as right shift – very efficient.**
- **Unsigned Divisor – one cycle. (Division call costs 35 cycles)**
- **Signed Divisor – more expensive. (Could cast to unsigned?)**

  x / 2^n = ((x<0) ? (x+2^n-1) : x) >> n // Consider –1/4 = 0!

- **Example: signed int / 16**

```
R3 = [I1];              // load divisor
CC = R3 <  0;           // check if negative
R1 =   15;              // add 2^n-1 to divisor
R2 = R3 + R1;
IF CC R3 = R2 ;         // if divisor negative use addition result
R3 >>>=  4;             // to the divide as a shift
```

- **Ensure compiler has visibility. Divisor must be unambiguous.**

Beware Hidden Division

- **Division can be created by For loops.**
- **Sometimes the compiler will calculate number of iterations.**

- **for (I = start; I < finish; I += step)**

  compiler plants code to calculate:

  iterations = (finish-start) / step

Division Trick 1 – Multiply by Reciprocal.

```
   float recip_NUM_SAMPS = 1.0/NUM_SAMPS;
for (i=0; i<NC; i++) {
    for (j=0; j<NC; j++) {
      float sum = 0.0;
      for (k=0; k<NUM_SAMPS; k++)
          sum += Input[i*NC + k] * Input[j*NC + k];
      Cover[i*NC + j] = sum / NUM_SAMPS;
                      = sum * recip_NUM_SAMPS ;
   }
}
```

- **Replace Division by Multiplication by Reciprocal**
  - helps when divisor is locally constant
  - answer may be slightly different - is this OK?

—Analog Devices Confidential Information—

Use the laws of Algebra

- **Original customer benchmark compares ratios coded as:**

if (X/Y > A/B)

Recode as:

if (X * B > A * Y)

Another way to lose divisions!

- **Problem: possible overflow in fixed point.**
- **The compiler does not know anything about the real data precision. The programmer must decide. For instance two 12 bit precision inputs are quite safe. (24 bits max on multiplication.)**

Replace Conditionals with Min,Max,Abs.

Simple bounded decrement

```
k = k-1;
if (k <  -1)
   k = -1;
```

Programming “trick”

```
k = max (k-1, -1);
```

Avoid jump instruction latencies and simplifying control flow helps optimisation.

```
R0 +=  -1;
R1 =   -1;
R0 = MAX (R1,R0);
```

The compiler will often do this automatically for you, but not always in 16 bit cases.

BF ISA Note: Min and Max are for signed values only.

—Analog Devices Confidential Information—

ANALOG DEVICES

Removing Conditionals 2

- **Pipelined Architecture Problem:**

```
sum = 0;
for (I=0; I<NN; I++) {
 if ( KeyArray[val1][10-k+I] == '1' )
    sum = sum + buffer[I+10]*64;
 else
    sum = sum -  buffer[I+10]*64;
}
```

- **Better Solution removes conditional branch. Multiplication is fast: let KeyArray hold +64 or -64**

```
sum = 0;
for (I=0; I<NN; I++)
  sum += buffer[I+10] * KeyArray[val1][10-k+I];
```

- **Compiler is not able to make this kind of global change**

ANALOG DEVICES

Removing conditionals 3

- **Duplicate small loops rather than have a conditional in a small loop.**

- **Example**

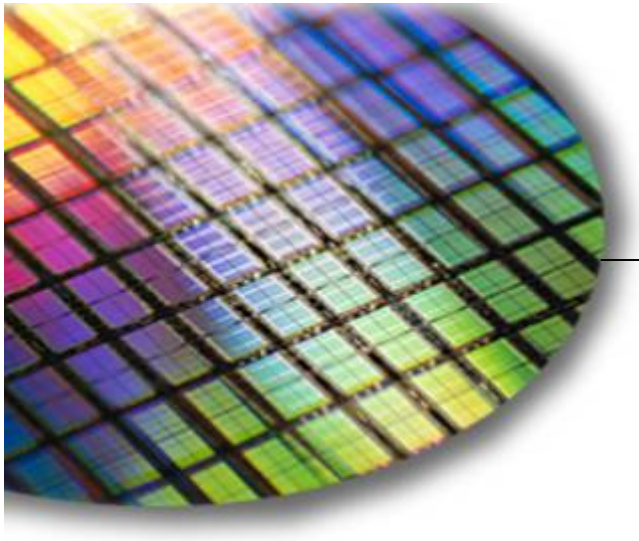
```
for {
    if  { ….. } else {…..}
}

=>  if {
        for {…..}
    } else {
        for  {…..}
    }
```


ANALOG DEVICES

Removing Conditionals 4
Predicated Instruction Support

- **The blackfin predicated instruction support takes the form of:**
- **IF (CC) reg = reg.**
- **Much faster than a conditional branch. (1 cycle) but limited.**
- **Help the compiler to see the opportunity.**
- **For instance – consider speculative execution.**
  - **if (A) X = EXPR1 else X = EXPR2;**
  - **X = EXPR1; IF (!A) X = EXPR2;**
  - **Or X=EXPR1; Y=EXPR2; if (!A) X=Y;**

ANALOG DEVICES



The World Leader in High Performance Signal Processing Solutions

Loops

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

The inner loop

- **The optimizer focuses on the inner loop because this is where most programs spend most of their time.**
- **Considered a good trade off to slow down loop prologue and epilogue to speed up loop.**
- **Make sure your program spends most of its time in the inner loop.**

Allow the optimizer to unroll loops

- **The optimizer "works by unrolling loops".**
  - **Vectorization**
  - **Software pipelining**
- **Do not unroll loops yourself.**
- **Avoid loop carried dependencies.**
- **Avoid aliases.**
- **Do not rotate loops yourself.**

Software Pipelining

- **What is software pipelining?**
  - Technique used to schedule loops and functional units efficiently.
  - Reorganizing the loop in such a way that each iteration of software-pipelined code is made from instructions of different iterations of the original loop

Simple Dot Product:
load, multiply, accumulate

```
CYCLE    1    2    3    4    5    6 .....100
                   F1   M1   A1
                        F2   M2   A2
                             F3   M3   A3
                                  F4   M4   A4
                                       ..................
```

  - The pipeline gives more instructions to be done per cycle.

Effects of Vectorization and Software Pipelining on Blackfin

- **Simple code generation: 1 iteration in 4 instructions**

```
LSETUP …;
R0.L = W[I1++]
R1.L = W[I0++];
A1+= R0.L*R1.L;
```

- **Vectorized and unrolled once: 2 iterations in 2 instructions**

```
R0 = [I1++]
R1 = [I0++]
A1+= R0.H*R1.H, A0+= R0.L*R1.L (IS)
```

- **Software pipeline: 2 iterations in 1 instruction**

```
            R0.L = W[I1++] || R0.H= W[I0++];
            LSETUP (._P1L2 , ._P1L3-8) LC0=P1;
.align 8;
._P1L2:
            A1+= R0.H*R0.H, A0+= R0.L*R0.H (IS) || R0.L = W[I1++] || R0.H= W[I0++];
._P1L3:
            A1+= R0.H*R0.H, A0+= R0.L*R0.H (IS);
```


ANALOG DEVICES

Do not unroll inner loops yourself

- **Good - compiler unrolls to use both compute blocks.**

```
for (i = 0; i < n; ++i)
   c[i] = b[i] + a[i];
```

- **Bad - compiler leaves on a single compute block.**

```
for (i = 0; i < n; i+=2) {
   xb = b[i]; yb = b[i+1];
   xa = a[i]; ya = a[i+1];
   xc = xa + xb; yc = ya + yb;
   c[i] = xc; c[i+1] = yc;
}
```

- OK to unroll outer loops.

ANALOG DEVICES

Avoid loop carried dependencies

- **Bad: Scalar dependency.**

```
for (i = 0; i < n; ++i)
    x = a[i] - x;
```

Value used form previous iteration. So iterations cannot be overlapped.

- **Bad: Array dependency.**

```
for (i = 0; i < n; ++i)
    a[i] = b[i] * a[c[i]];
```

Value may be from previous iteration. So iterations cannot be overlapped.

ANALOG DEVICES

Resolvable dependencies

- **Good: A Reduction.**

```
for (i = 0; i < n; ++i)
  x = x + a[i];
```

Operation is associative. Iterations can be reordered to calculate the same result.

- **Good: Induction variables.**

```
for (i = 0; i < n; ++i)
  a[i+4] = b[i] * a[i];
```

Addresses vary by a fixed amount on each iteration. Compiler can see there is no data dependence.

ANALOG DEVICES

Avoid aliases

- **Is there a loop carried dependence in this loop?**

```
void fn(int a[], int b[], int n) {
  for (i = 0; i < n; ++i)
    a[i] = b[i];
}
```

Yes, if a and b point at the same array.

- **Write your code so they do not point at the same array.**
- **-ipa switch may help compiler find out this is so.**

ANALOG DEVICES

Do not rotate loops yourself

- **A common DSP idiom. To rotate loops so loads can be executed at same time as computation.**
- **Introduces loop carried dependencies.**
- **Makes code less easy to read.**
- **The compiler can do it for itself.**
- **Just don't do it.**

ANALOG DEVICES

The original loop (good)

```
float ss(float *a, float *b, int n){

float sum = 0.0f;
int i;
for (i = 0; i < n; i++)
{
sum += a[i] + b[i];
}
return sum;
}
```

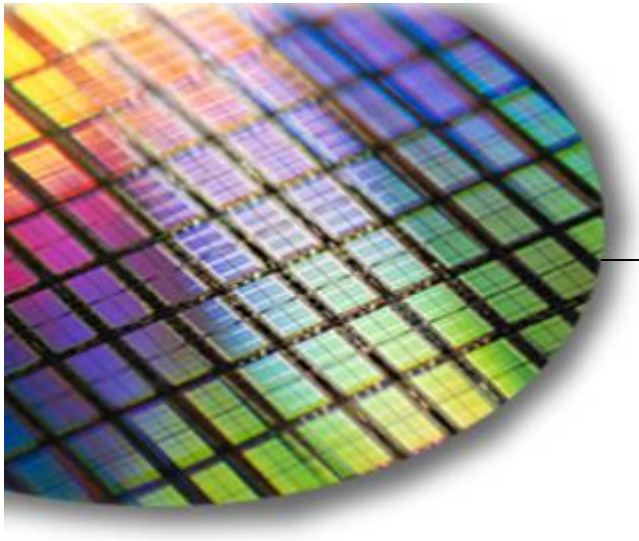
A rotated loop (bad)

```
float ss(float *a, float *b, int n
  ) {

  float ta, tb , sum = 0.0f;
  int i = 0;
  ta = a[i]; tb = b[i];
  for (i = 1; i < n; i++) {
    sum += ta + tb;
    ta = a[i]; tb = b[i];
  }
  sum += ta + tb;
  return sum;
}
```


ANALOG DEVICES

Experiment with Loop structure

- **Unify inner and outer Loops.**
  - May make loop too complex, but optimiser is better focused.
- **Loop Inversion. - reverse nested loop order.**
- **Unify sequential loops –**
  - reduce memory accesses – can be crucial when dealing with external memory.



The World Leader in High Performance Signal Processing Solutions

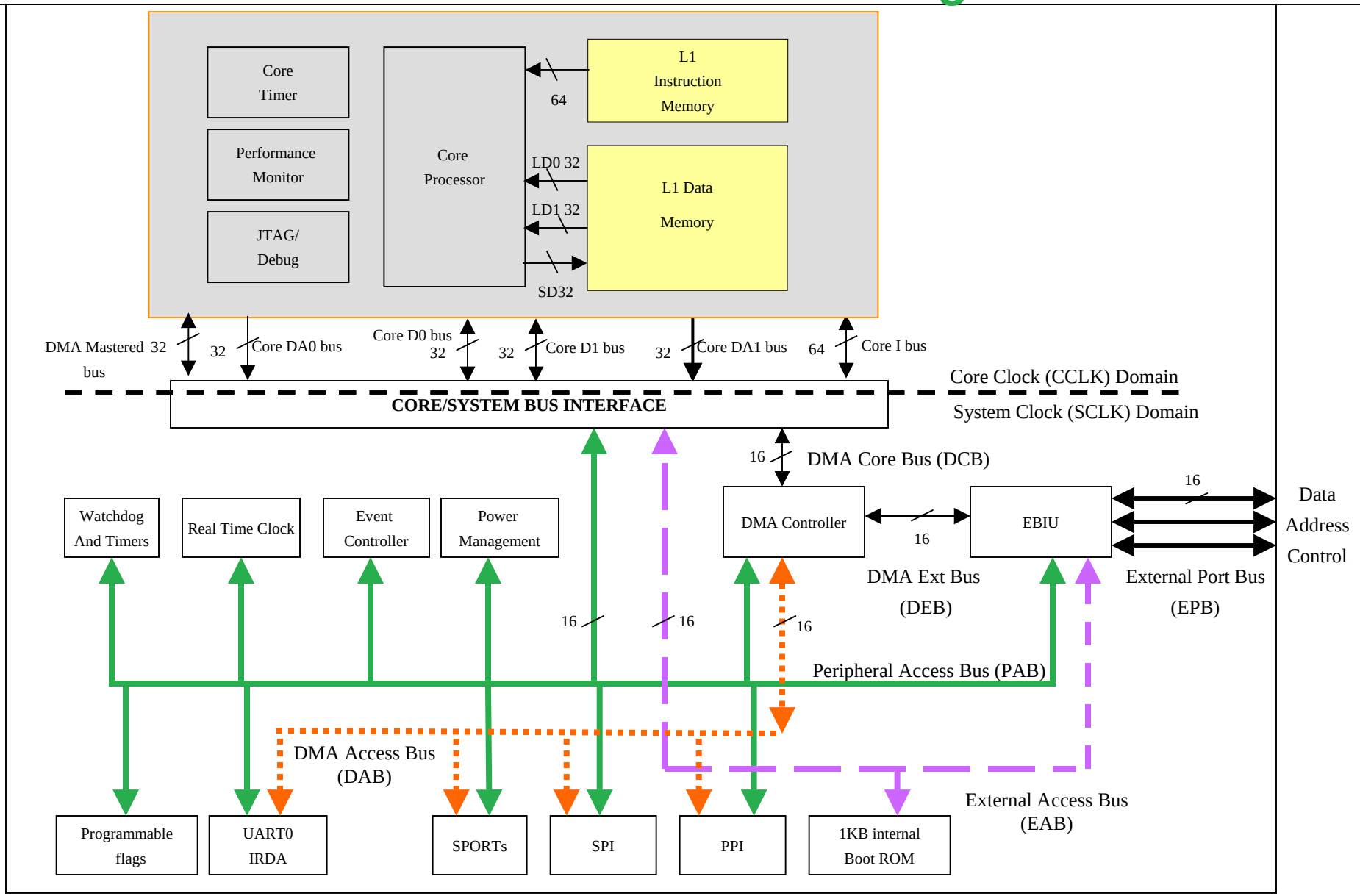
Section 6

Blackfin ADSP-BF533 Memory

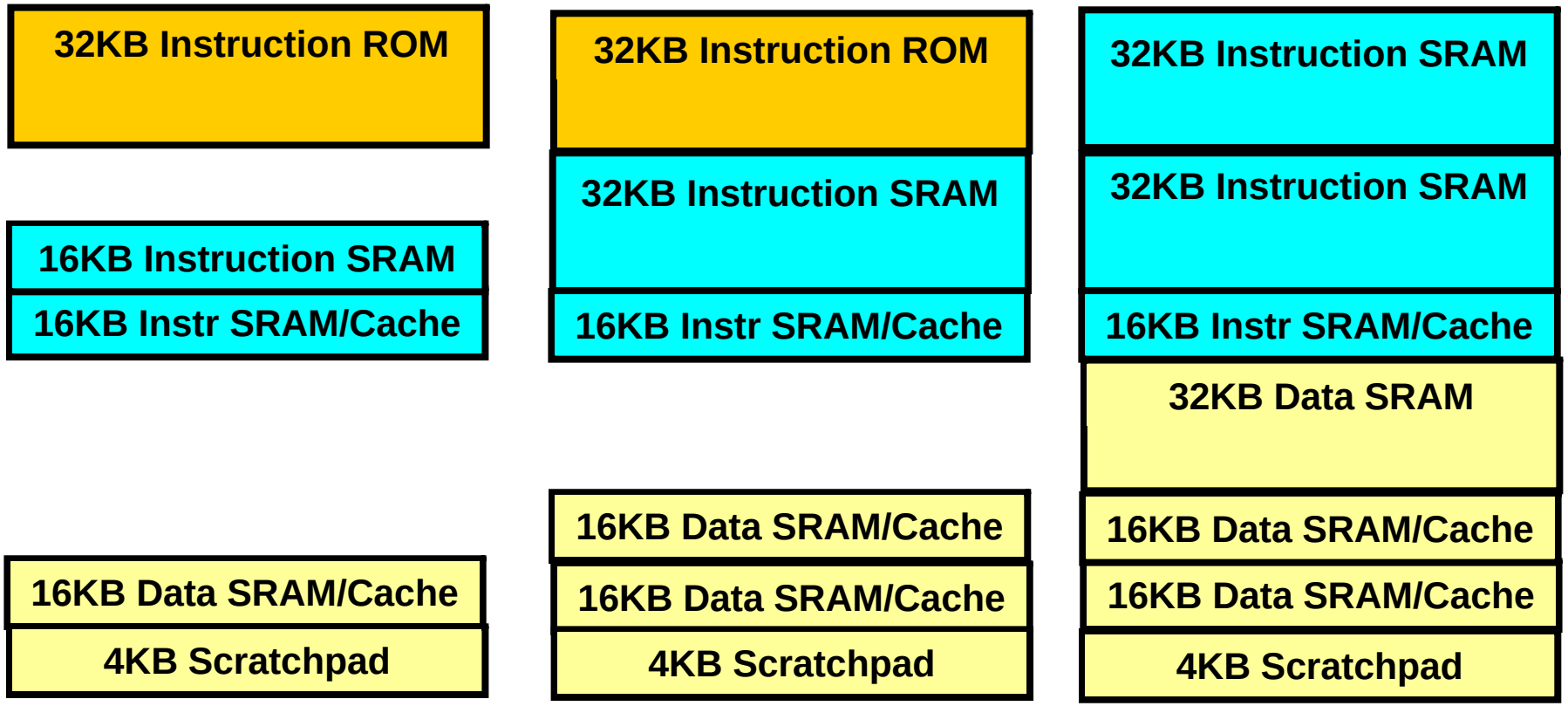
ADI Confidential Information – Not for external distribution

ANALOG DEVICES

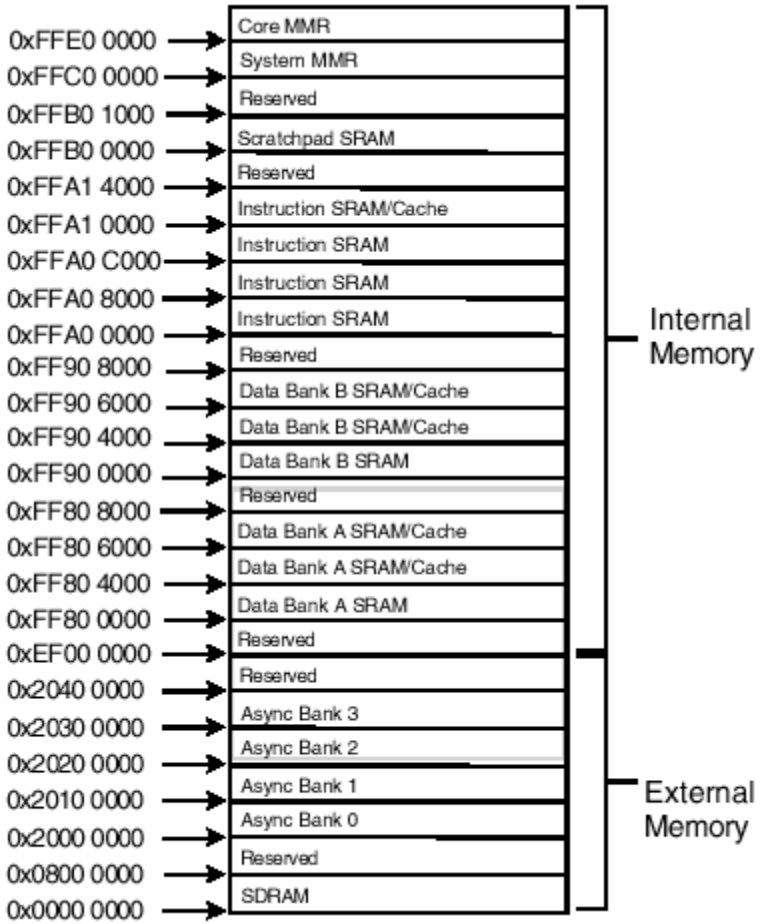
ADSP-BF533 Block Diagram



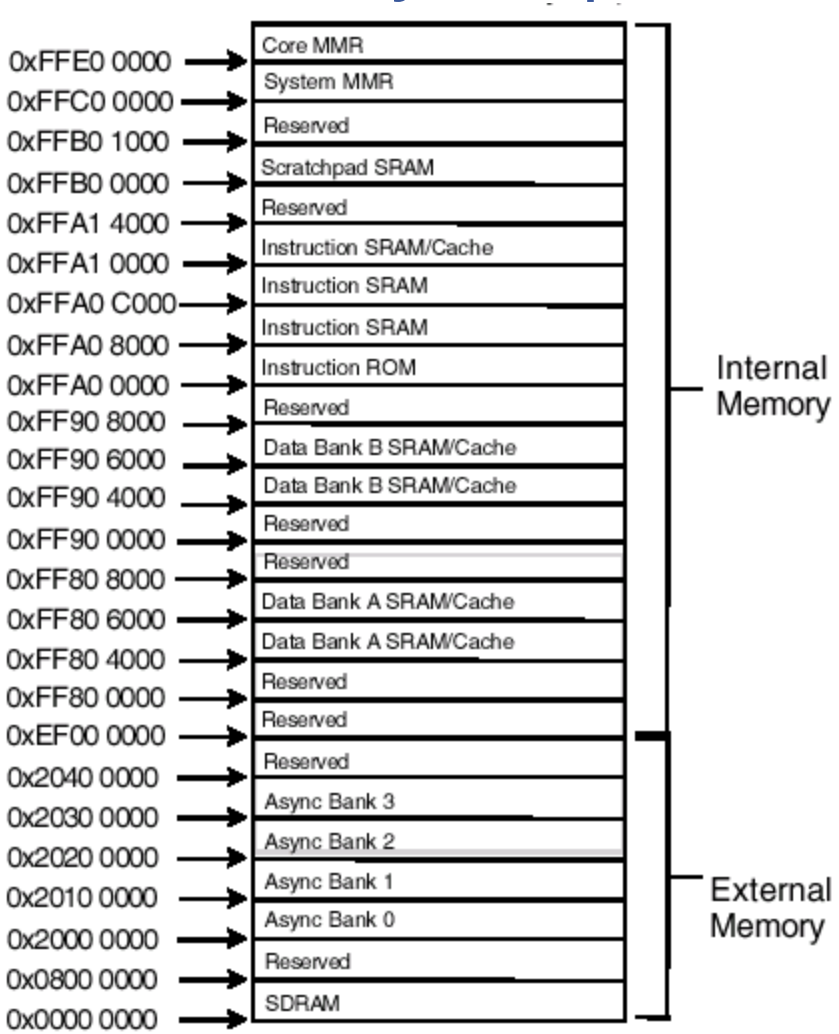
Blackfin Internal SRAM



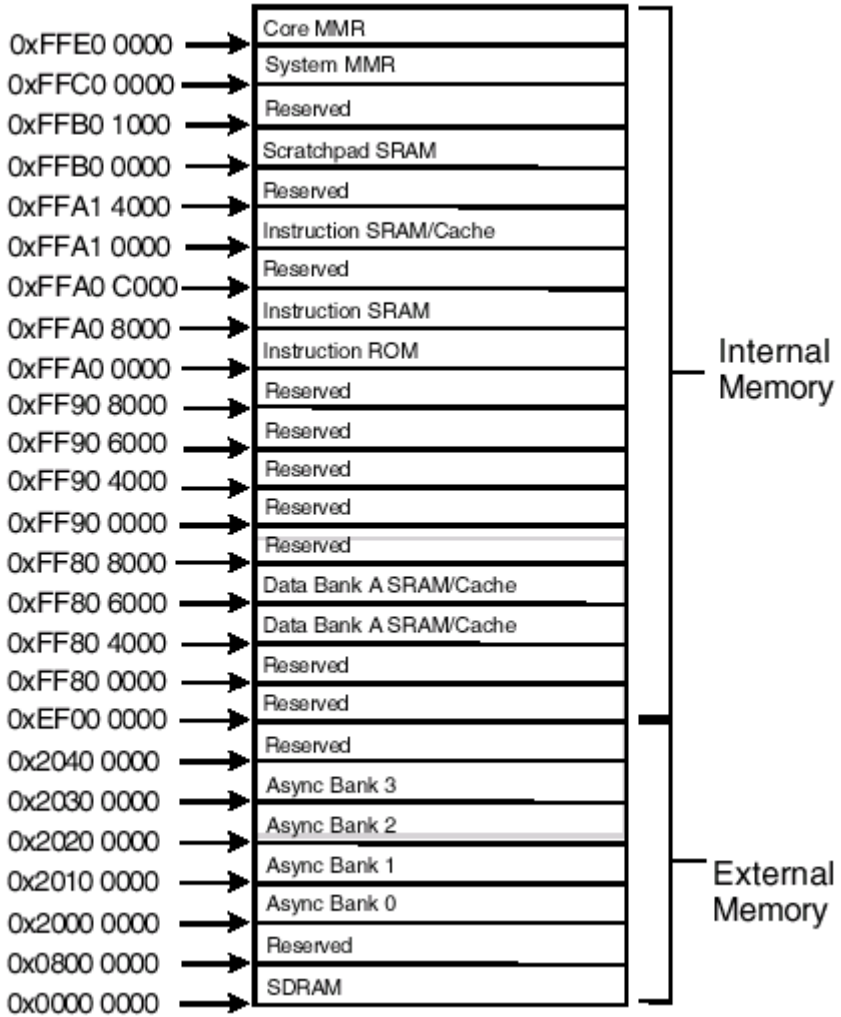
ADSP-BF533 Memory Map



ADSP-BF532 Memory Map

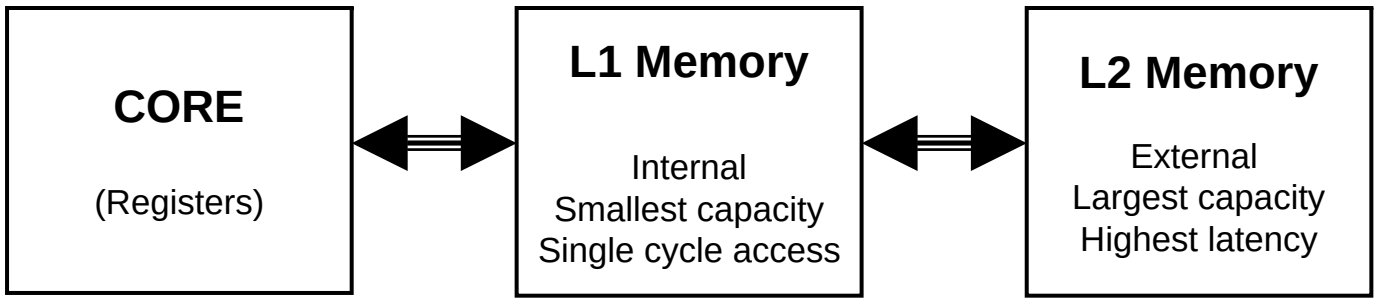


ADSP-BF531 Memory Map

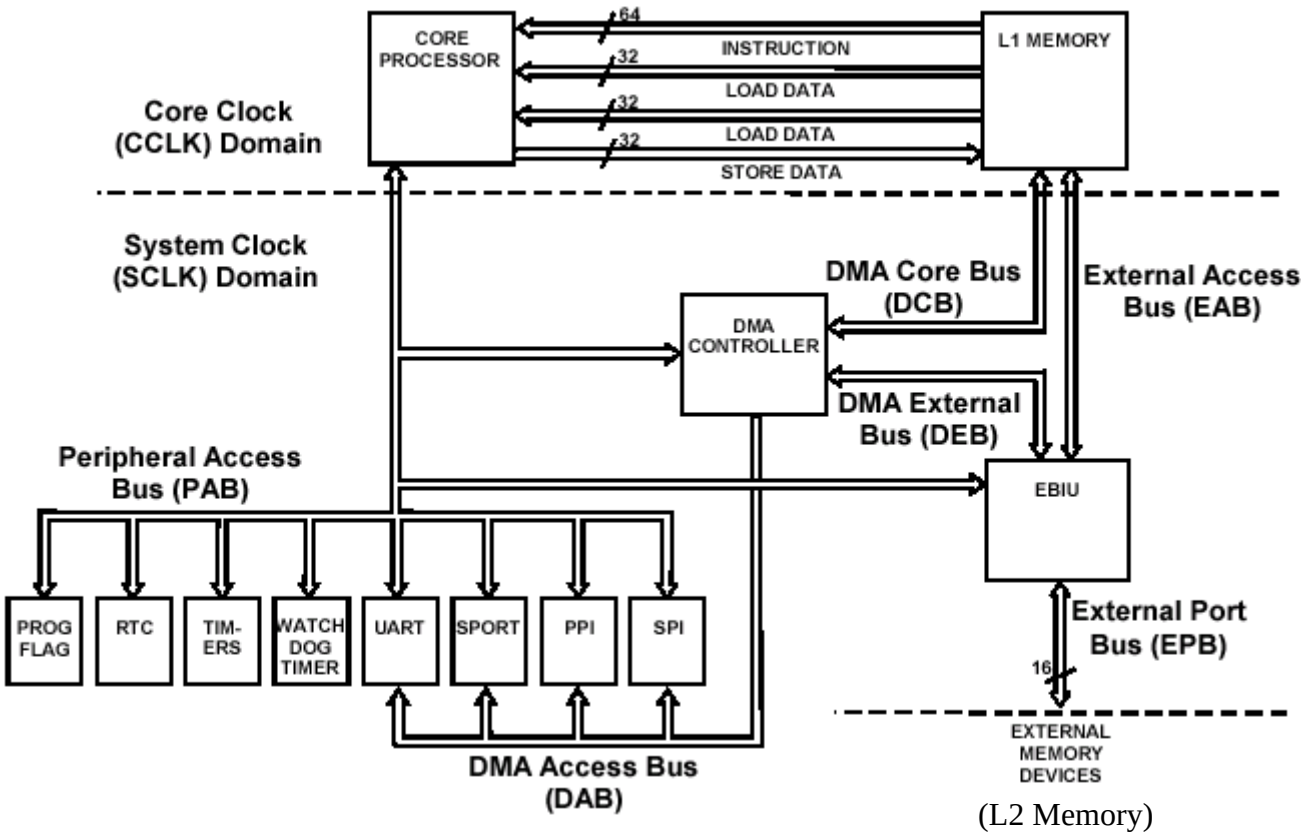


Memory Hierarchy on the BF533

- **As processor speeds increase (300Mhz – 1 GHz), it becomes increasingly difficult to have large memories running at full speed.**
- **The BF53x uses a *memory hierarchy* with a primary goal of achieving memory performance similar to that of the fastest memory (i.e. L1) with an overall cost close to that of the least expensive memory (i.e. L2)**



Internal Bus Structure of the ADSP-BF533



(L2 Memory)

Configurable Memory

- **The best system performance can be achieved when executing code or fetching data out of L1 memory**
- **Two methods can be used to fill the L1 memory – Caching and Dynamic Downloading – Blackfin® Processor Supports Both.**
  - **Micro-controllers have typically used the caching method, as they have large programs often residing in external memory and determinism is not as important.**
  - **DSPs have typically used Dynamic Downloading as they need direct control over which code runs in the fastest memory.**
- **Blackfin® Processor allows the programmer to choose one or both methods to optimize system performance.**

Why Do Blackfin® Processors Have Cache?

- **To allow users to take advantage of single cycle memory without having to specifically move instructions and or data "manually"**
  - **L2 memory can be used to hold large programs and data sets**
  - **The paths to and from L1 memory are optimized to perform with cache enabled**
- **Automatically optimizes code that reuses recently used or nearby data**

Internal L1 Memory: Smallest capacity Single cycle access	External L2 Memory: Largest capacity Highest latency

Configurable L1 Memory Selections

L1 Instruction	L1 Data A	L1 Data B (BF533 and BF532 only)	L1 Data Scratchpad
Cache	Cache	Cache	SRAM
Cache	Cache	SRAM	SRAM
Cache	SRAM	SRAM	SRAM
SRAM	Cache	Cache	SRAM
SRAM	Cache	SRAM	SRAM
SRAM	SRAM	SRAM	SRAM

Using instruction cache will improve performance for most applications

Data Cache may or may not improve performance

Max bandwidth into L1 memory is available with cache enabled

Trade-offs must be made on code control and peak short-term performance

ANALOG DEVICES

Core MMR L1 Memory Registers

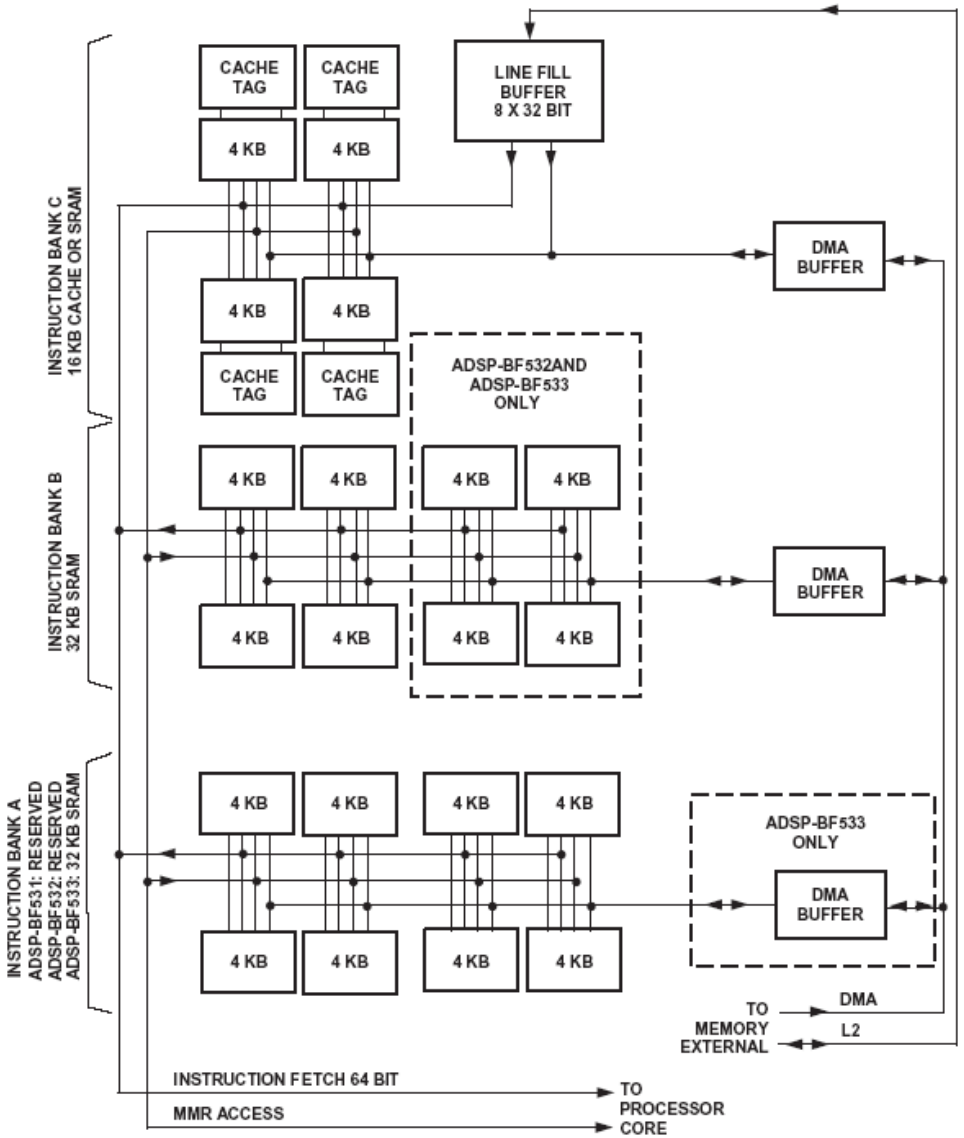
- **General Control**
  - **IMEM_CONTROL (Instruction Memory)**
  - **DMEM_CONTROL (Data Memory)**
- **Cache and Protection Properties (n=0 to 15)**
  - **ICPLB_DATAn, ICPLB_ADDRn**
  - **DCPLB_DATAn, ICPLB_ADDRn**
- **Test Functionality**
  - **ITEST_COMMAND, ITEST_DATA**
  - **DTEST_COMMAND, DTEST_DATA**

BF533 L1 Instruction Memory

Instruction Bank C
BF531, BF532, BF533:
16KB SRAM/CACHE

Instruction Bank B
BF531: 16KB SRAM
BF532: 32KB SRAM
BF533: 32KB SRAM

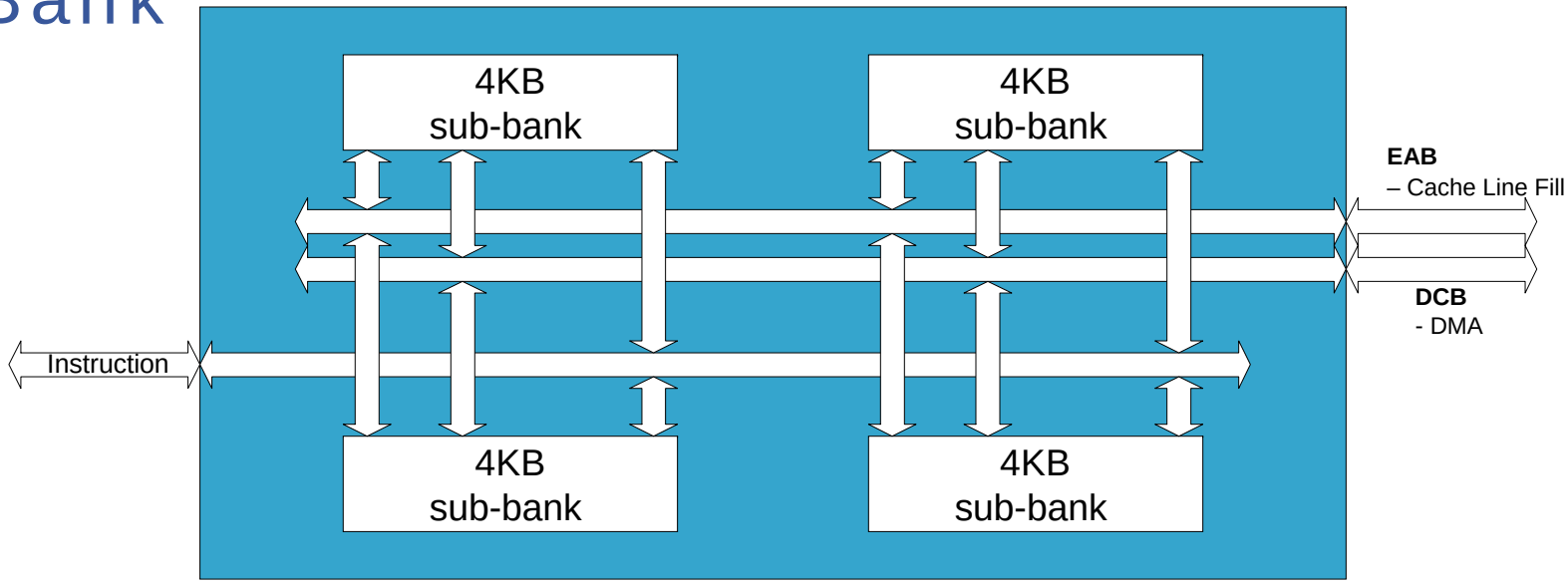
Instruction Bank A
BF531: 32KB ROM
BF532: 32KB ROM
BF533: 32KB SRAM



—Analog Devices Confidential Information—

ANALOG DEVICES

L1 Instruction Memory 16KB Configurable Bank



16 KB SRAM

- Four 4KB single-ported sub-banks
- Allows simultaneous core and DMA accesses to different banks

16 KB cache

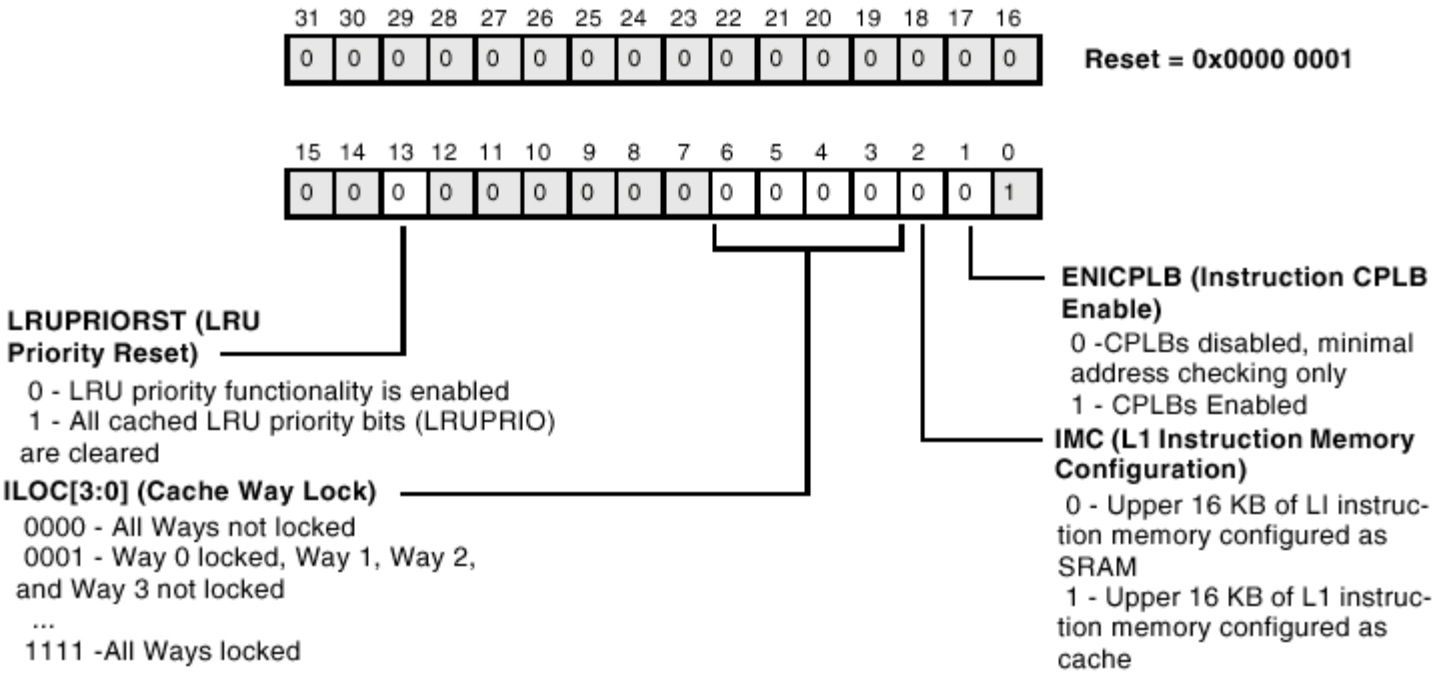
- 4-way set associative with arbitrary locking of ways and lines
- LRU replacement
- No DMA access

Features of L1 Instruction Memory Unit

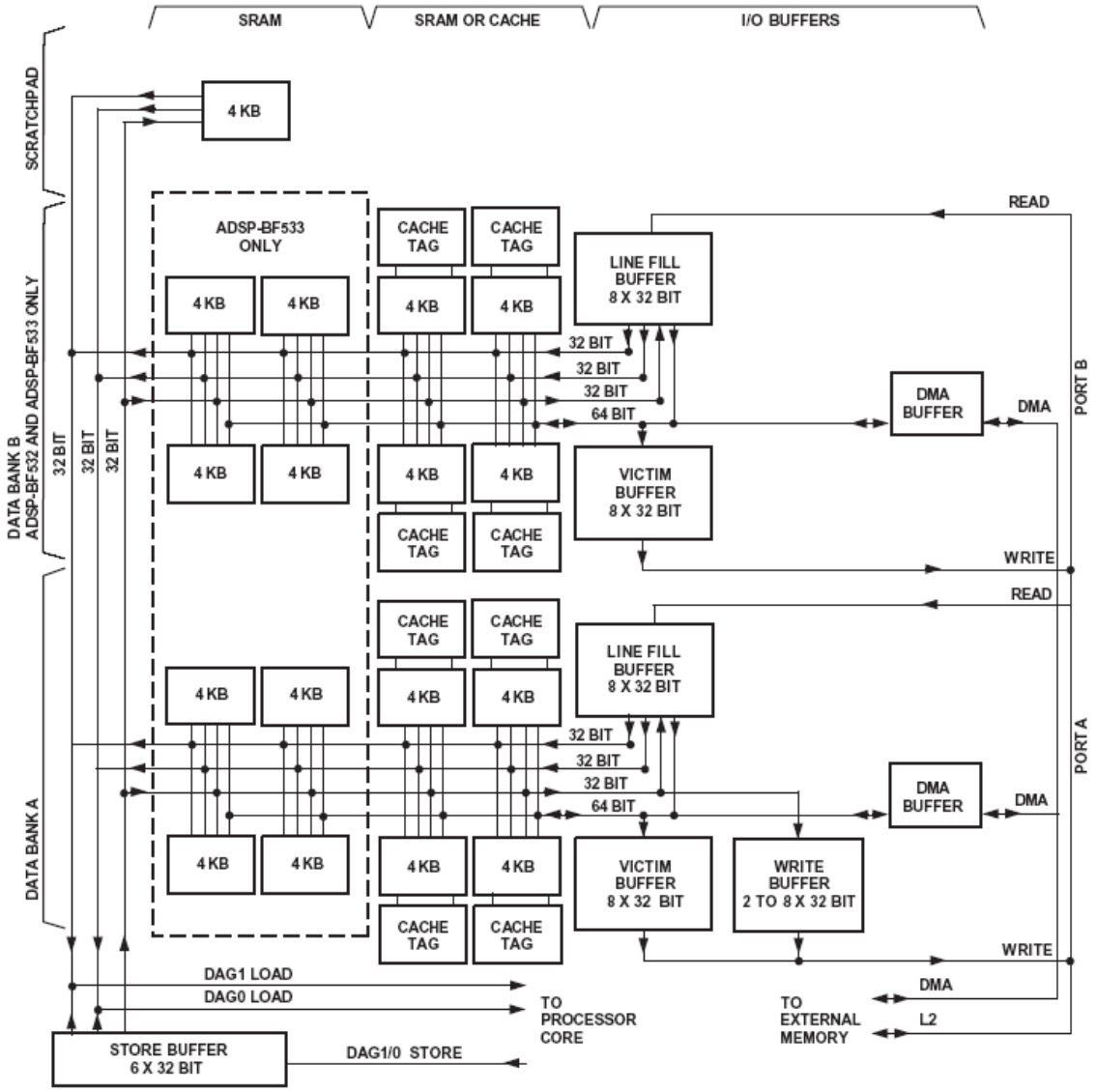
- **Instruction Alignment Unit: handles alignment of 16-, 32-, and 64-bit instructions that are to be sent to the execution unit.**
- **Cacheability and Protection Look-aside Buffer (CPLB): Provides cacheability control and protection during instruction memory accesses.**
- **256-bit cache Line Fill Buffer: uses four 64-bit word burst transfers to copy cache lines from external memory.**
- **Memory test interface: Provides software with indirect access to tag and data memory arrays.**

L1 Instruction Memory Control Register

IMEM_CONTROL



BF533 L1 Data Memory



Victim Buffers:
Victimized Write-Back Cached Data to external memory

Write Buffer:
Write-Through and Non-cached Data to external memory

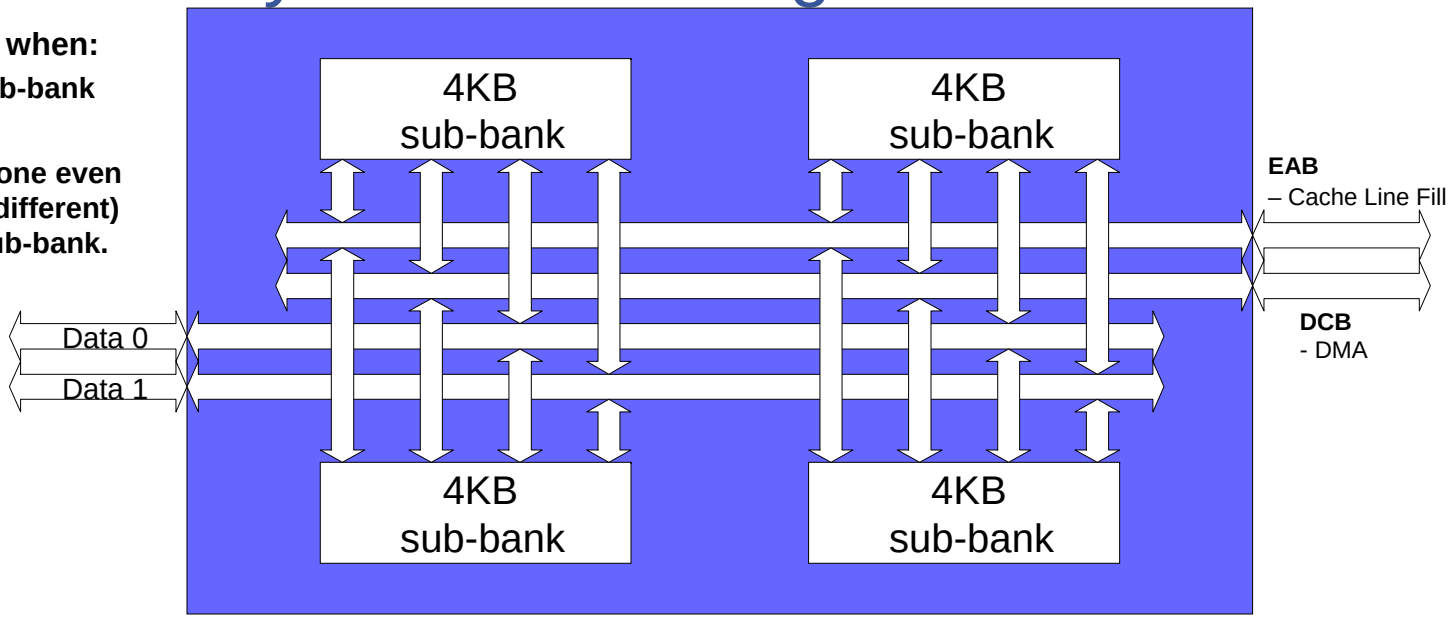
L1 Data Memory 16KB Configurable Bank

Block is Multi-ported when:

Accessing different sub-bank

OR

Accessing one odd and one even access (Addr bit 2 different) within the same sub-bank.



- When Used as SRAM
  - Allows simultaneous dual DAG and DMA access
- When Used as Cache
  - Each bank is 2-way set-associative
  - No DMA access
  - Allows simultaneous dual DAG access

BF533 L1 Data Memory

Sub-Bank	Data Bank A	Data Bank B	
1	0xFF80 0000	0xFF90 0000	SRAM
2	0xFF80 1000	0xFF90 1000	
3	0xFF80 2000	0xFF90 2000	
4	0xFF80 3000	0xFF90 3000	
5	0xFF80 4000	0xFF90 4000	CONFIGURABLE
6	0xFF80 5000	0xFF90 5000	
7	0xFF80 6000	0xFF90 6000	
8	0xFF80 7000	0xFF90 7000	

L1 configurable data memory can be:

- Both banks A & B as SRAM
- Bank A as cache, bank B as SRAM
- Both banks as cache

ANALOG DEVICES

BF532 L1 Data Memory

Sub-Bank	Data Bank A	Data Bank B	
1	0xFF80 0000	0xFF90 0000	SRAM
2	0xFF80 1000	0xFF90 1000	
3	0xFF80 2000	0xFF90 2000	
4	0xFF80 3000	0xFF90 3000	
5	0xFF80 4000	0xFF90 4000	CONFIGURABLE
6	0xFF80 5000	0xFF90 5000	
7	0xFF80 6000	0xFF90 6000	
8	0xFF80 7000	0xFF90 7000	

L1 configurable data memory can be:

- Both banks A & B as SRAM
- Bank A as cache, bank B as SRAM
- Both banks as cache

ANALOG DEVICES

BF531 L1 Data Memory

Sub-Bank	Data Bank A	Data Bank B	
1	0xFF80 0000	0xFF90 0000	SRAM
2	0xFF80 1000	0xFF90 1000	
3	0xFF80 2000	0xFF90 2000	
4	0xFF80 3000	0xFF90 3000	
5	0xFF80 4000	0xFF90 4000	CONFIGURABLE
6	0xFF80 5000	0xFF90 5000	
7	0xFF80 6000	0xFF90 6000	
8	0xFF80 7000	0xFF90 7000	

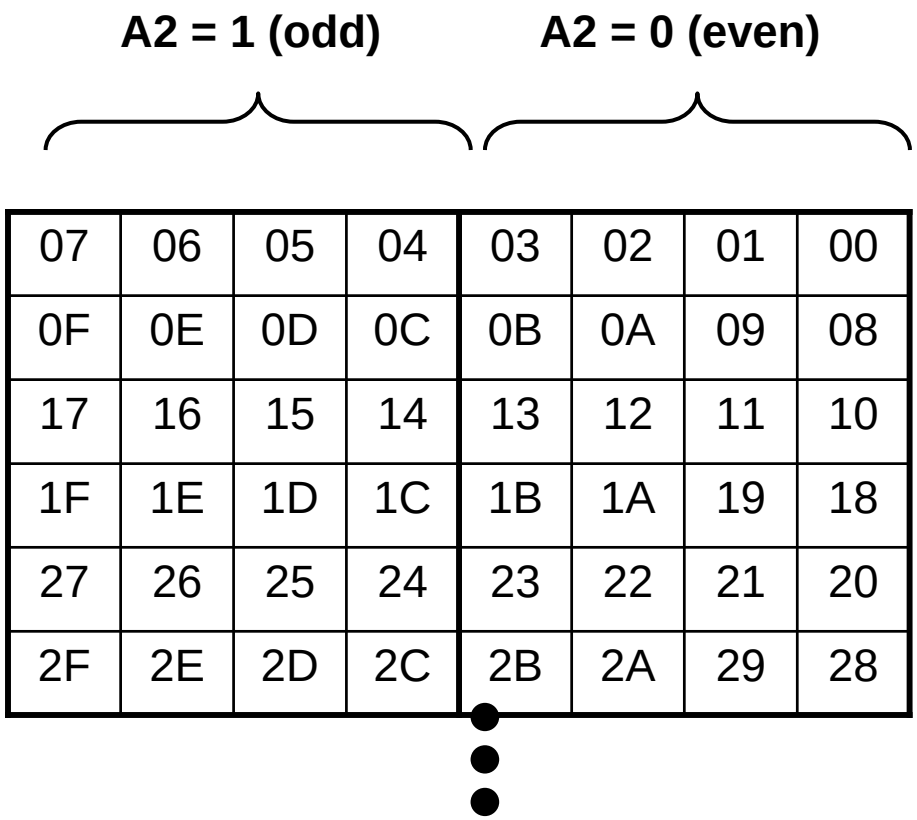
L1 configurable data memory can be:

- Bank A as SRAM
- Bank A as Cache

L1 Data Memory SRAM Addressing

- **Both DAG units can access Data Banks A & B**
- **If an address conflict is detected Data Bank priority is as follows:**
  1. **System DMA (highest priority)**
  2. **DAG Unit 0**
  3. **DAG Unit 1 (lowest priority)**
- **Parallel DAG accesses can occur to the same Data Bank as long as the references are to different sub-banks OR they access 2 words of different 32-bit address polarity (Address bit 2 is different).**

Dual Access to Same Sub-Bank



A dual access to an odd and even (quad address) location can be performed in a single cycle

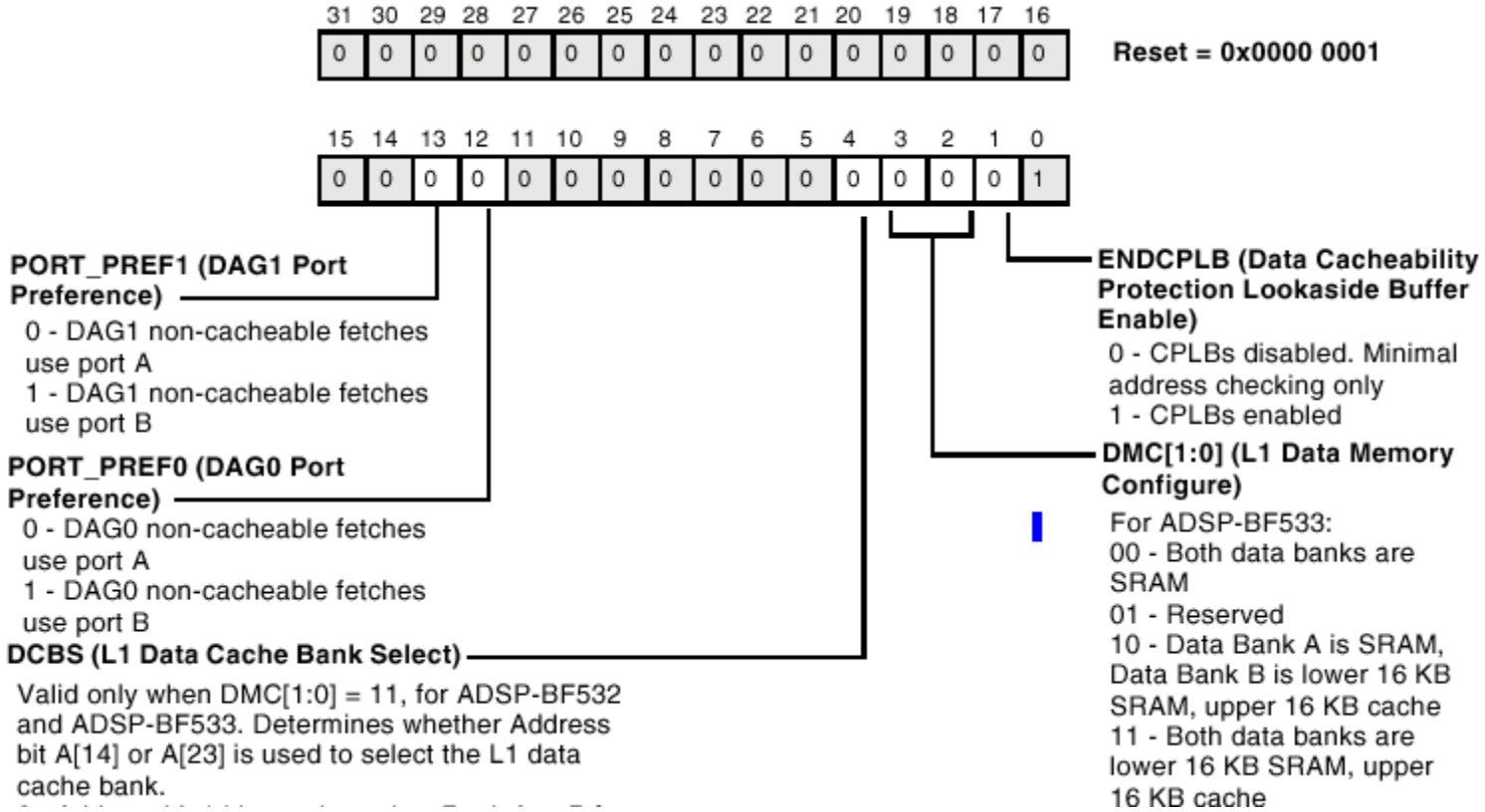
A dual access to two odd or two even locations will result in an extra cycle (1 stall) of delay

L1 Scratchpad Memory

- **Dedicated 4KB Block of Data SRAM**
- **Operates at CCLK rate**
- **Can not be configured as Cache**
- **Can not be accessed by DMA**
- **Typical Use is for User and Supervisor stacks to do fast context switching during interrupt handling.**

L1 Data Memory Control Register

DMEM_CONTROL

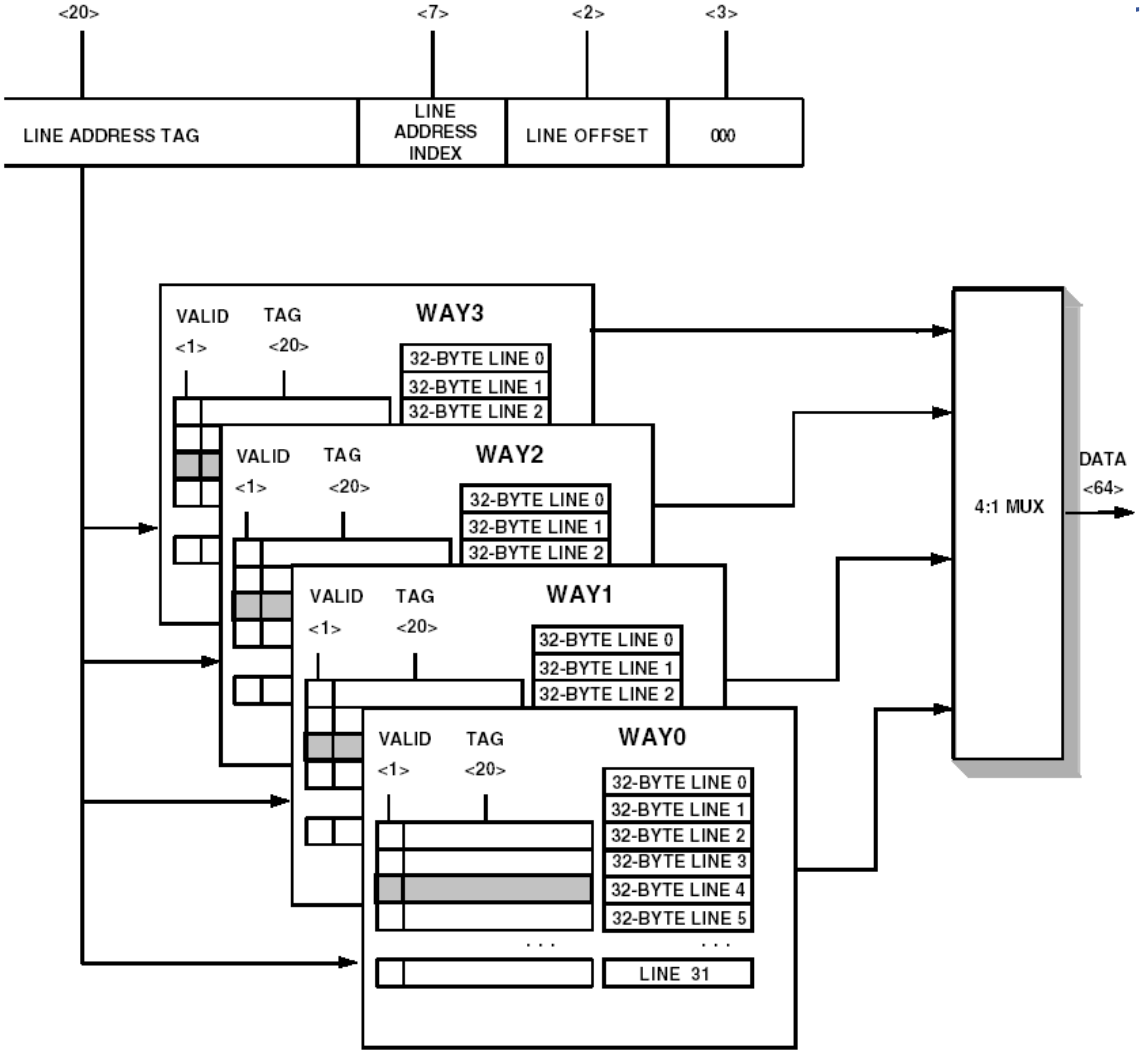


Cache Mode

What is Cache?

- **In a hierarchical memory system, *cache* is the first level of memory reached once the address leaves the core (i.e L1)**
  - **If the instruction/data word (8, 16, 32, or 64 bits) that corresponds to the address is in the cache, there is a *cache hit* and the word is forwarded to the core from the cache.**
  - **If the word that corresponds to the address is not in the cache, there is a *cache miss*. This causes a fetch of a fixed size block (which contains the requested word) from the main memory.**
    - The Blackfin allows the user to specify which regions (i.e. *pages*) of main memory are cacheable and which are not through the use of CPLBs (more on this later).
      - If a page is cacheable, the block (i.e. *cache line* containing 32 bytes) is stored in the cache after the requested word is forwarded to the core
      - If a page is non-cacheable, the requested word is simply forwarded to the core

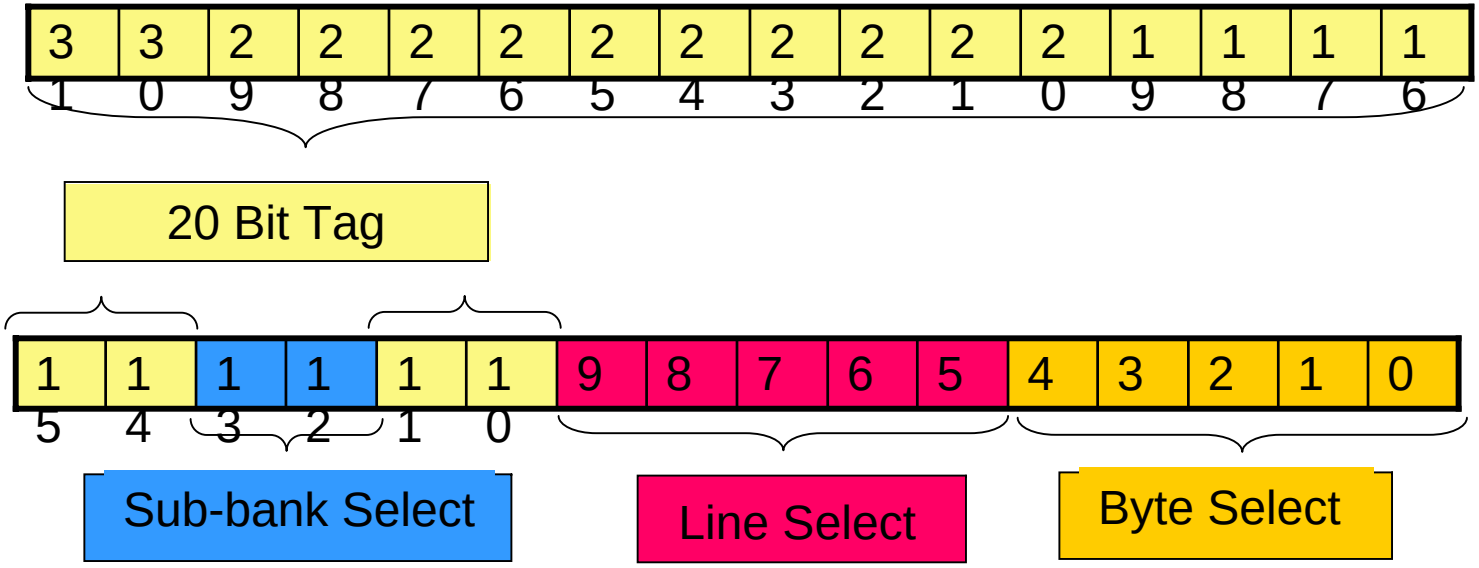
ADSP-BF533 Instruction Cache



- **Cache Line:**
  - **A 32 byte contiguous block of memory**
- **Set:**
  - **A group of cache lines in the cache**
    - Selected by Line Address Index
- **Way:**
  - **One of several places in a set that a cache line can be stored**
    - 1 of 4 for Instructions
    - 1 of 2 for Data
- **Cache Tag:**
  - **Upper address bits stored with cache line. Used to ID the specific address in main memory that the cached line represents**

Instruction Cache Placement Based On Address

- Four 4KB sub-banks (16KB total)
- Each sub-bank has 4-ways (1KB for each way)
- Each way has 32 lines
- Each line is 32 bytes



ANALOG DEVICES

Cache Hits and Misses

- A cache hit occurs when the address for an instruction fetch request from the core matches a valid entry in the cache.
- A cache hit is determined by comparing the upper 18 bits, and bits 11 and 10 of the instruction fetch address to the address tags of valid lines currently stored in a cache set.
- Only valid cache lines (i.e. cache lines with their valid bits set) are included in the address tag compare operation.
- When a cache hit occurs, the target 64-bit instruction word is sent to the instruction alignment unit where it is stored in one of two 64-bit instruction buffers.
- When a cache miss occurs, the instruction memory unit generates a cache line-fill access to retrieve the missing cache line from external memory to the core.

Instruction Fill from L2 Memory

- Cache Off
  - –64 bits

- Cache On
  - –Burst Cache Line fill (32-bytes)

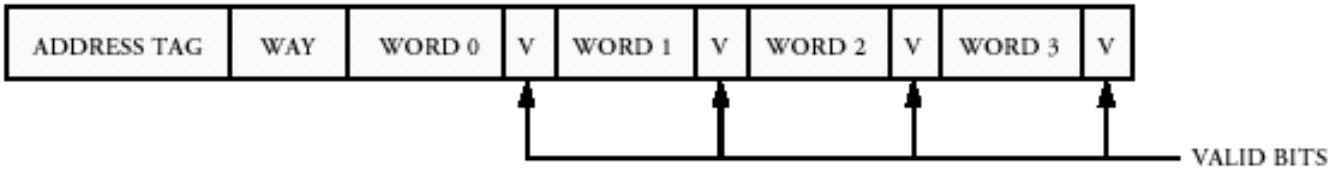
Cache Line Fills

- A cache line fill consists of fetching 32 bytes of data from memory external to the core (i.e. L2 memory).
- A line read data transfer consists of a four 64-bit word read burst.
- The instruction memory unit requests the target instruction word first; once it has returned the target word the IMU requests the next three words in sequential address order and wrap around if necessary.

Target Word	Fetching Order for Next Three Words
WD0	WD0, WD1, WD2, WD3
WD1	WD1, WD2, WD3, WD0
WD2	WD2, WD3, WD0, WD1
WD3	WD3, WD0, WD1, WD2

Cache Line-Fill Buffer

- **The cache line-fill buffer allows the core to access the data from the new cache line as the line is being retrieved from external memory, rather than having to wait until the line has been completely written to the 4KB memory block.**
- **The line-fill buffer organization is shown below:**



- **The line-fill buffer is also used to support non-cacheable accesses*. A non-cacheable access consists of a single 64-bit transfer on the instruction memory unit's external read port.**

*** A non-cacheable access includes: external accesses when instruction memory is configured as SRAM, or accesses to non-cacheable pages**

Cache Line Replacement

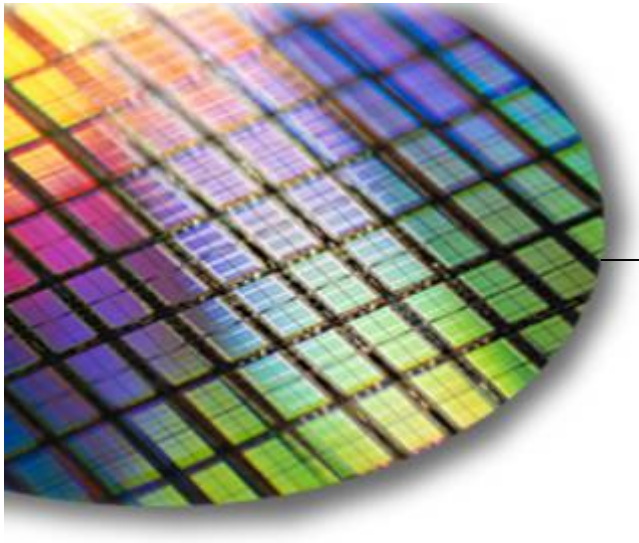
- **The cache line replacement unit first checks for invalid entries.**
- **If only a single invalid entry is found then that entry is selected for the new cache line. If multiple invalid entries are found the replacement entry for the new cache line is selected based on the following priority:**
  - **way 0 first**
  - **way 1 next**
  - **way 2 next**
  - **way 3 last**
- **When no invalid entries are found, the cache replacement logic uses a 6-bit LRU algorithm to select the entry for the new cache line.**
- **For instruction cache the LRUPRIO bit is also considered.**

Instruction Cache "Locking By Line" (LRUPRIO)

- **Prevents the Cached Line from being replaced**
- **CPLB_LRUPRIO bits in the ICPLB_DATAx register define the priority for that page.**
- **The Cache line importance level (LRUPRIO) is saved in the TAG and used by the replacement policy logic.**
- **Cache Line Replacement policy with LRUPRIO**
  - **No invalid entries:**
    - A high priority will replace a low priority or a high priority if all 4-ways contain high priority lines.
    - LRU (least recently used) policy is used to determine which one of the lines that have the same priority will be replaced.
- **Setting the IMEM_CONTROL: LRUPRIORST bit clears all LRUPRIO bits in the TAGs.**

Instruction Cache Locking By Way

- **Each 4KB way of the instruction cache can be locked individually to ensure placement of performance-critical code.**
- **Controlled by the ILOC<3:0> bits in the IMEM_CONTROL register.**



The World Leader in High Performance Signal Processing Solutions

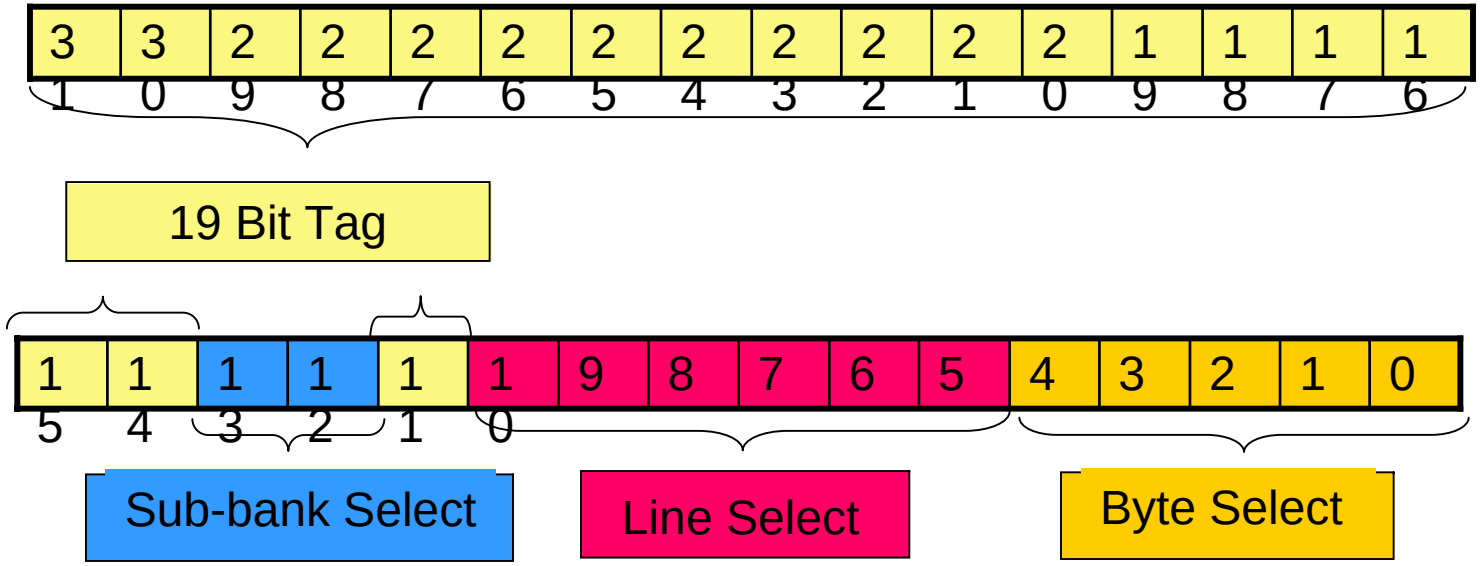
Data Cache Mode

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Data Cache Placement Based On Address

- Four 4KB sub-banks (16KB total)
- Each sub-bank has 2-ways (2KB for each way)
- Each way has 64 lines
- Each line is 32 bytes
- If Both Data Bank A and B are set for Cache, bit 14 or 23 is used to determine which Data Bank.



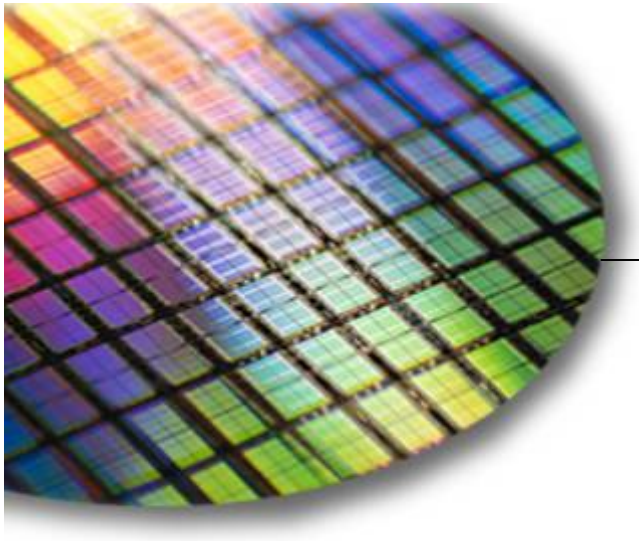
ANALOG DEVICES

Data Cache Definitions

- **Write Through:**
  - **A cache write policy where write data is written to the cache line and to the source memory.**
- **Write Back:**
  - **A cache write policy where write data is written only to the cache line. The modified cache line is written to source memory only when it is replaced.**
- **Dirty/Clean (Applies to Write Back Mode only):**
  - **State of cache line indicating whether the data in the cache has changed since it was copied from source memory**
- **Performance trade-off required between write through and write back to determine the best policy to use for an application.**

Data Cache Victim Buffer

- **The victim buffer is used to read a dirty cache line either being flushed or replaced by a cache line fill and then to initiate a burst write operation on the bus to perform the line copyback to the system.**
- **The processor can continue running without having to wait for the data to be written back to L2 memory.**
- **The victim buffer is comprised of a 4-deep FIFO each 64-bits in width (similar to the fill-buffer.)**
- **There is no data forwarding support from the victim buffer.**



The World Leader in High Performance Signal Processing Solutions

Cacheability Protection Lookaside Buffers (CPLBS)

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Memory Protection and Cache Properties

- **Memory Management Unit**
  - **Cacheability and Protection Look-Aside Buffers (CPLBs)**
  - **Cache/protection properties determined on a per memory page basis (1K, 4K, 1M, 4M byte sizes)**
  - **32 CPLBs total: 16 CPLBs for instruction memory, 16 CPLBs for data memory**
- **User/Supervisor Access Protection**
- **Read/Write Access Protection**
- **Cacheable or Non-Cacheable**

Using CPLBs

- **Cache enabled:**
  - **CPLB must be used to define cacheability properties**
    - **When CPLBS are enabled, a valid CPLB must exist before an access to a specific memory location is attempted. Otherwise, an exception will be generated.**
    - **User and Supervisor mode protection is available without using CPLBs.**

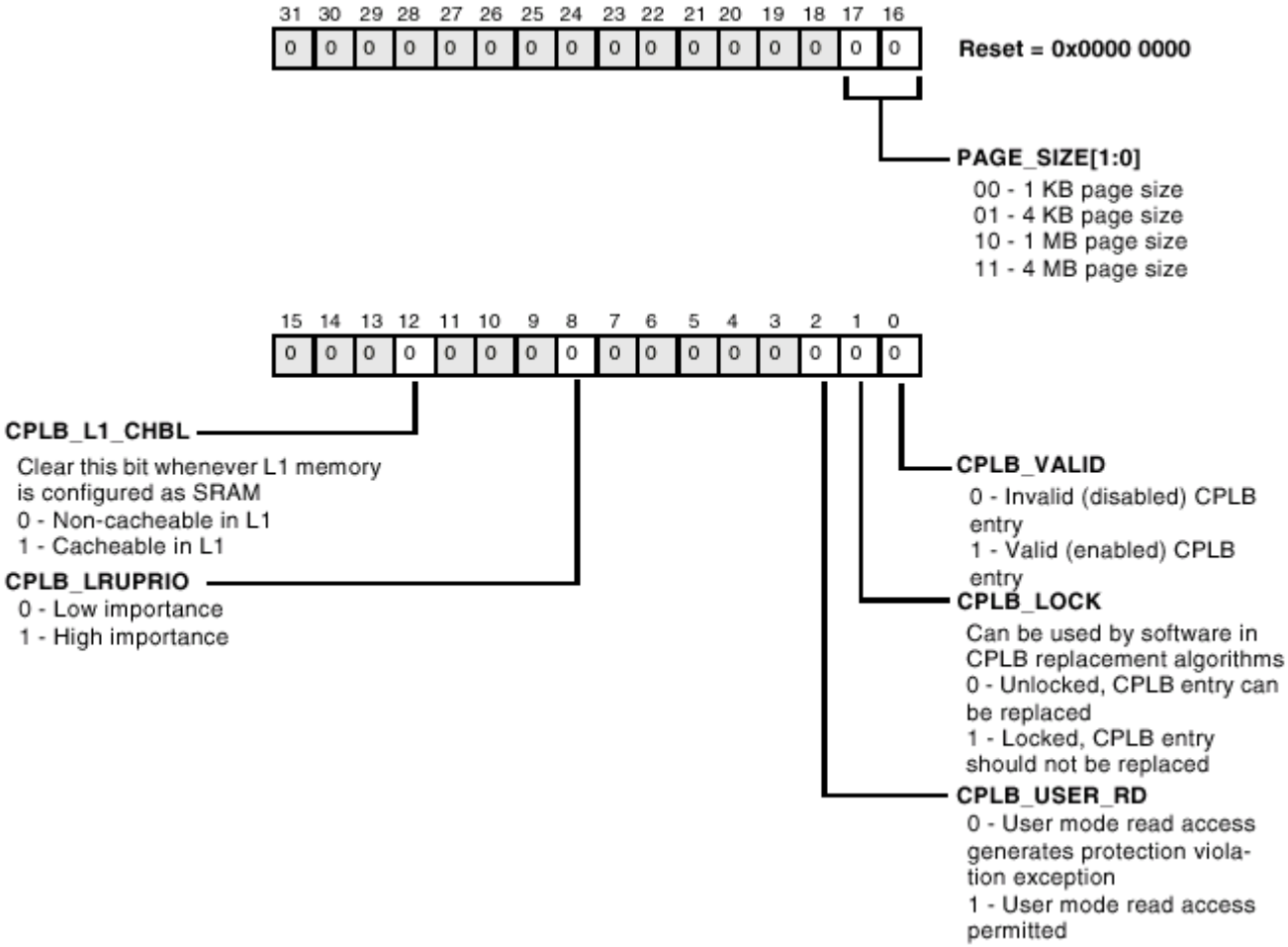
- **Cache disabled:**
  - **CPLBs can be used to protect pages of memory**

Cacheability Protection Lookaside Buffers (CPLBs)

- **Divide the entire Blackfin memory map into regions (i.e. pages) that have cacheability and protection properties.**
- **16 Pages in Instruction Memory plus 16 Pages in Data memory**
  - **Page sizes: 1KB, 4KB, 1MB, 4MB**
- **Each CPLB has 2 associated registers:**
  - **32bit Start Address: ICPLB_ADDRn, DCPLB_ADDRn**
  - **Cache/Protection Properties: ICPLB_DATAn, DCPLB_DATAn**

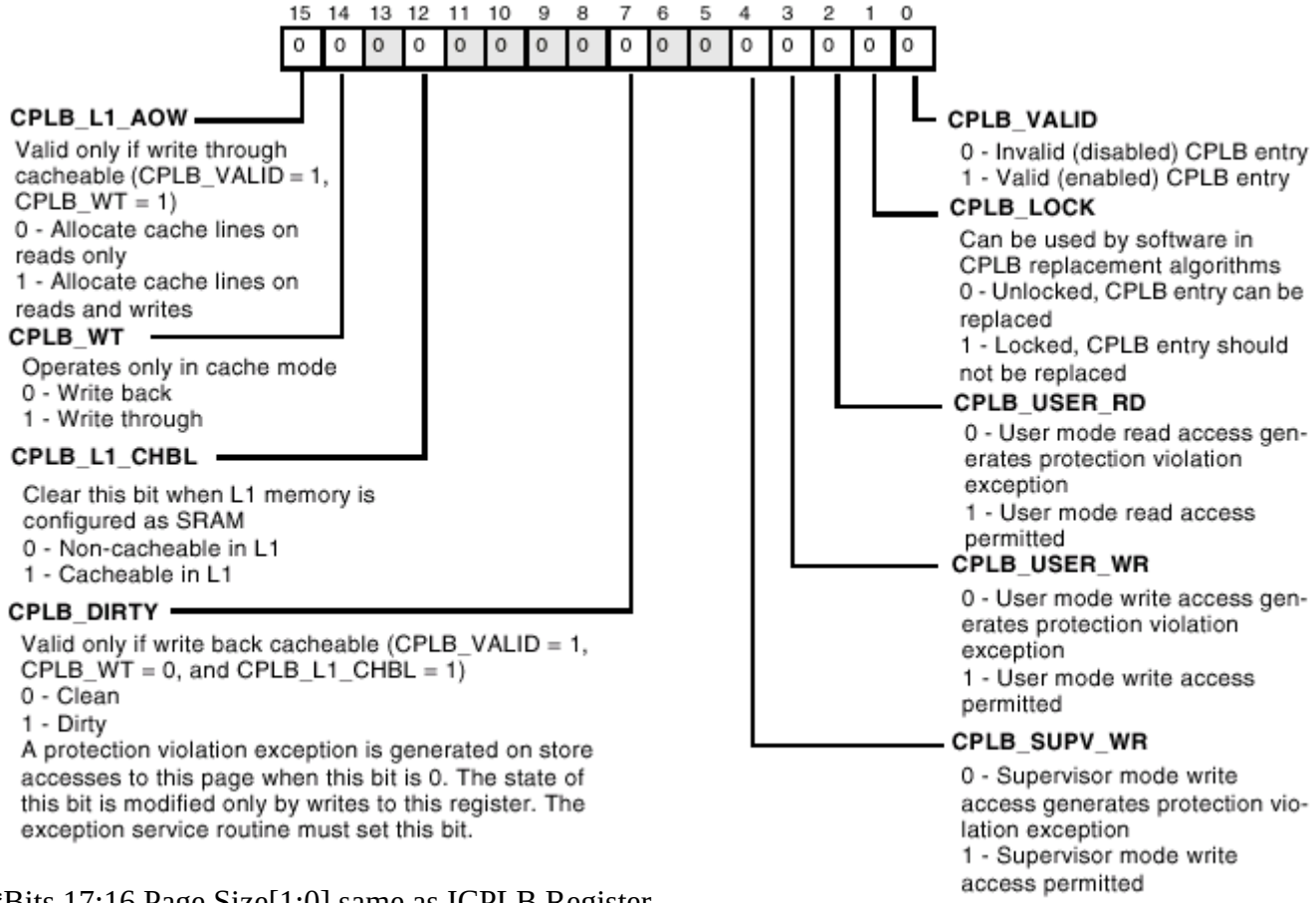
Note: “n” equals 15:0

ICPLB_DATAn Register



Note: “n” equals 15:0

DCPLB_Datan Register



*Bits 17:16 Page Size[1:0] same as ICPLB Register

Note: “n” equals 15:0

Example Protection Operation

- **Set up CPLBs to define regions and properties:**
  - **Default hardware CPLBs are present for MMRs and scratchpad memory.**
  - **CPLBs must be configured for L1 Data and L1 Instruction Memory as Non-Cacheable**
  - **Disable all memory other than the desired memory space.**
  - **Execute Code.**
- **If code tries to access memory that has been ‘disabled’ or protected, then a ‘memory protection violation’ occurs as an exception.**

Example CPLB Setup

Instruction CPLB setup

L1 Instruction: Non-cacheable
1MB page

SDRAM: Cacheable
Eight 4MB pages

Async: Non-cacheable
One 4MB page

Async: Cacheable
Two 4MB pages

Memory management handles exceptions and redefines external memory pages as required for external memory. Examples will be provided to customers.

Data CPLB setup

L1 Data: Non-cacheable
One 4MB page

SDRAM: Cacheable
Eight 4MB pages

Async: Non-cacheable
One 4 MB page

Async: Cacheable
One 4 MB page

Accessing the Cache Directly

- **Once L1 memory is configured as cache, it can't be accessed via DMA or from a core read.**
- **ITEST_COMMAND and ITEST_DATA memory mapped registers do allow direct access to Instruction Memory tags and lines.**
- **Analogous registers exist for Data Cache.**
- **Can be useful for invalidating cache lines directly.**

Data Cache Control Instructions

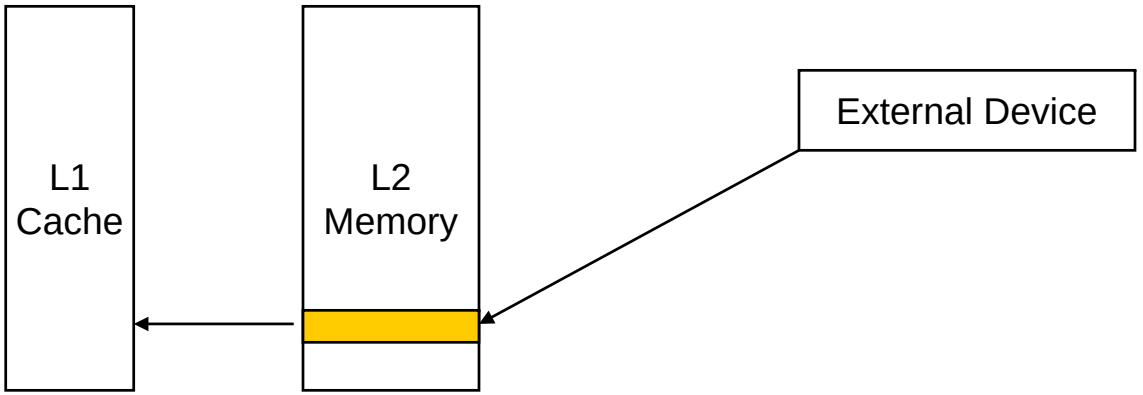
- **Prefetch: Causes data cache to prefetch line associated with address in P-register**
  - **Causes line to be fetched if it is not currently in the cache and the location is cacheable**
  - **Otherwise it behaves like a nop**
    - Prefetch [p2];
    - Prefetch [p2 ++]; // post increment by cache-line size
- **FLUSH: Causes data cache to synchronize specified cache line with higher levels of memory**
  - **If the line is dirty, it is written out and marked clean**
    - flush [p2];
    - flush [p2 ++]; // post increment by cache-line size
- **FLUSHINV: Causes data cache to invalidate a specific line in cache.**
  - **If the line is dirty, it is written out:**
    - flushinv [p2];
    - flushinv [p2 ++]; // post increment by cache-line size

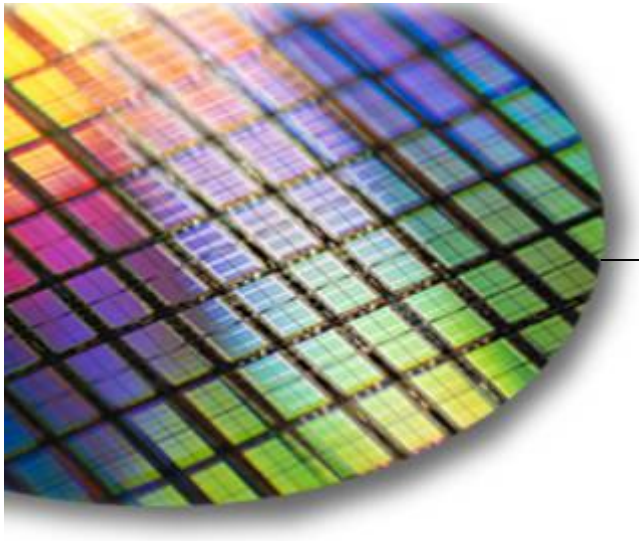
Instruction Cache Control Instructions

- **IFLUSH: Causes instruction cache to invalidate a specific line in cache.**
  - **iflush [p2];**
  - **iflush [p2 ++]; // post increment by cache-line size**

Coherency Considerations

- **Care must be taken when memory that is defined as "cacheable" is modified by outside source**
  - **DMA controller (data or descriptors)**
- **Cache is not aware of these changes so some mechanism must be setup**
  - **Simple memory polling will not work**
  - **Must Invalidate the cache before accessing the changed L2 memory.**





The World Leader in High Performance Signal Processing Solutions

Reference Material

Memory

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

Data Byte-Ordering

- **The ADSP-BF533 architecture supports little-endian byte-ordering**
- **For example, if the hex value 0x76543210 resides in register r0 and the pointer register p0 contains address 0x00ff0000, then the instruction "[p0] = r0;" would cause the data to be written to memory as shown below:**

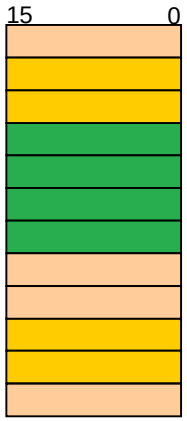
Byte Address	Data
0x00ff0000	0x10
0x00ff0001	0x32
0x00ff0002	0x54
0x00ff0003	0x76

- **When loading a byte, half-word, or word from memory to a register, the LSB (bit 0) of the data word is always loaded into the LSB of the destination register**

Instruction Packing

- **Instruction set tuned for compact code:**
  - **Multi-length instructions**
    - 16, 32, 64-bit opcodes
    - Limited multi-issue instructions
- **No memory alignment restrictions for code:**
  - **Transparent alignment H/W.**

Instruction Formats



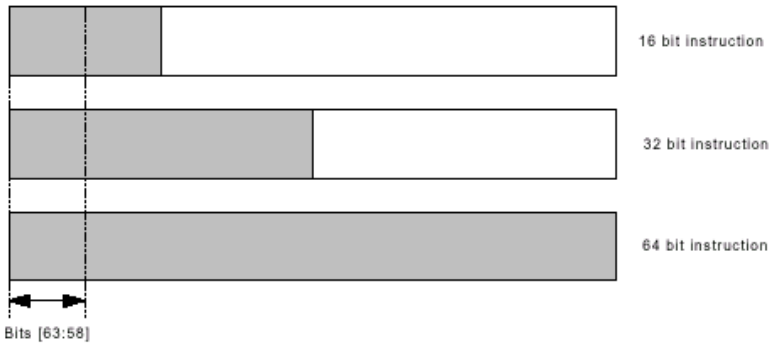
16-bit wide memory

No Memory Alignment Restrictions: Maximum Code Density and Minimum System Memory Cost

Instruction Fetching

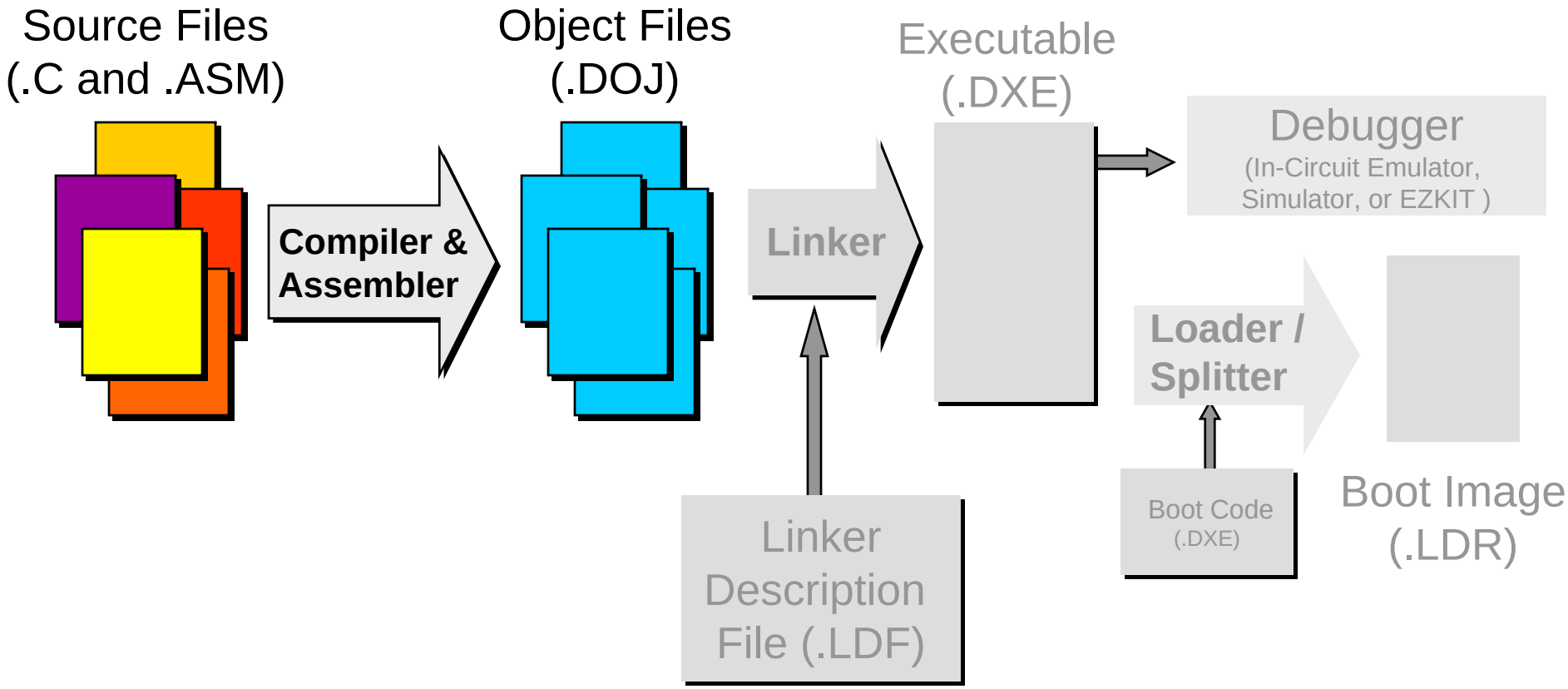
- 64-bit instruction line can fetch between 1 and 4 instructions

One 64-bit instruction			
One 32-bit instruction		One 32-bit instruction	
One 16-bit instruction	One 16-bit instruction	One 16-bit instruction	One 16-bit instruction
One 32-bit instruction		One 16-bit instruction	One 16-bit instruction



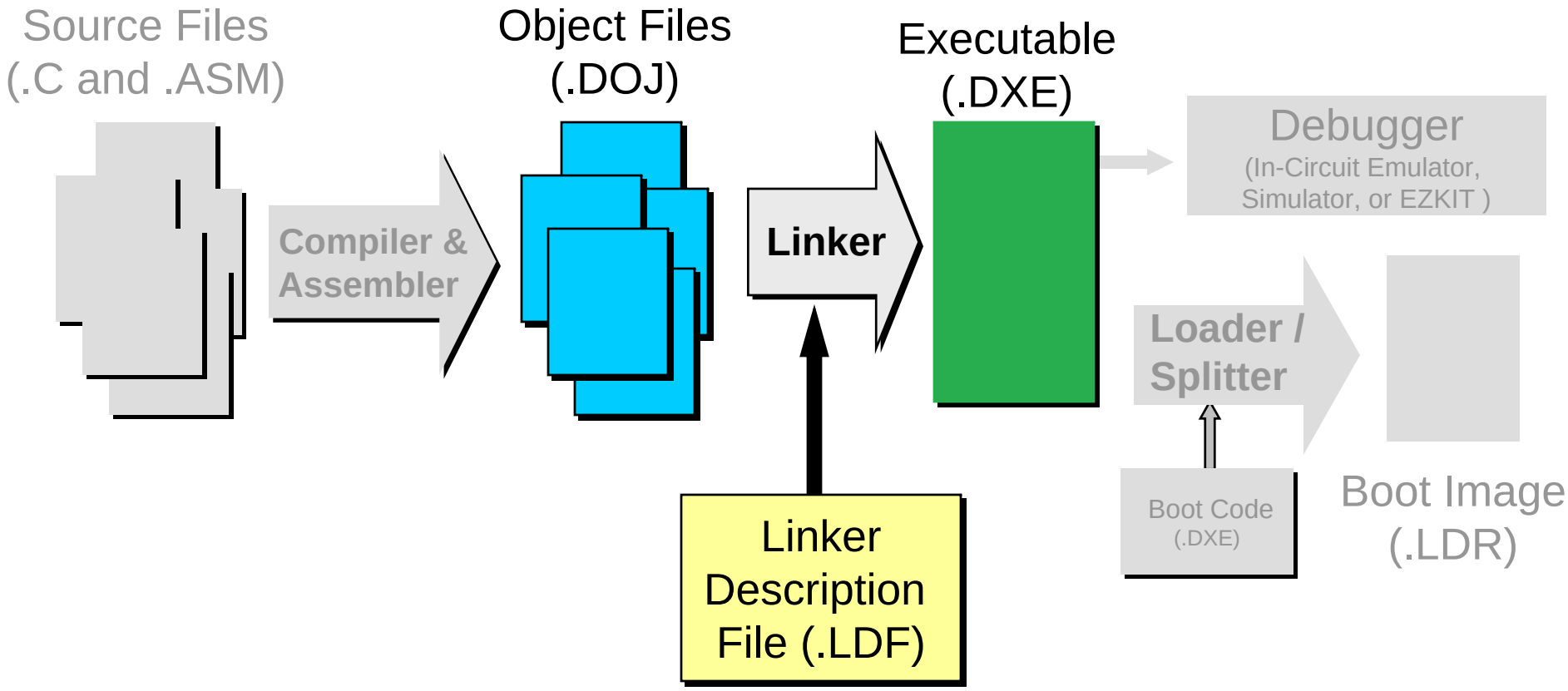
Linker Description File

Software Development Flow
Step 1- Compiling & Assembling



—Analog Devices Confidential Information—

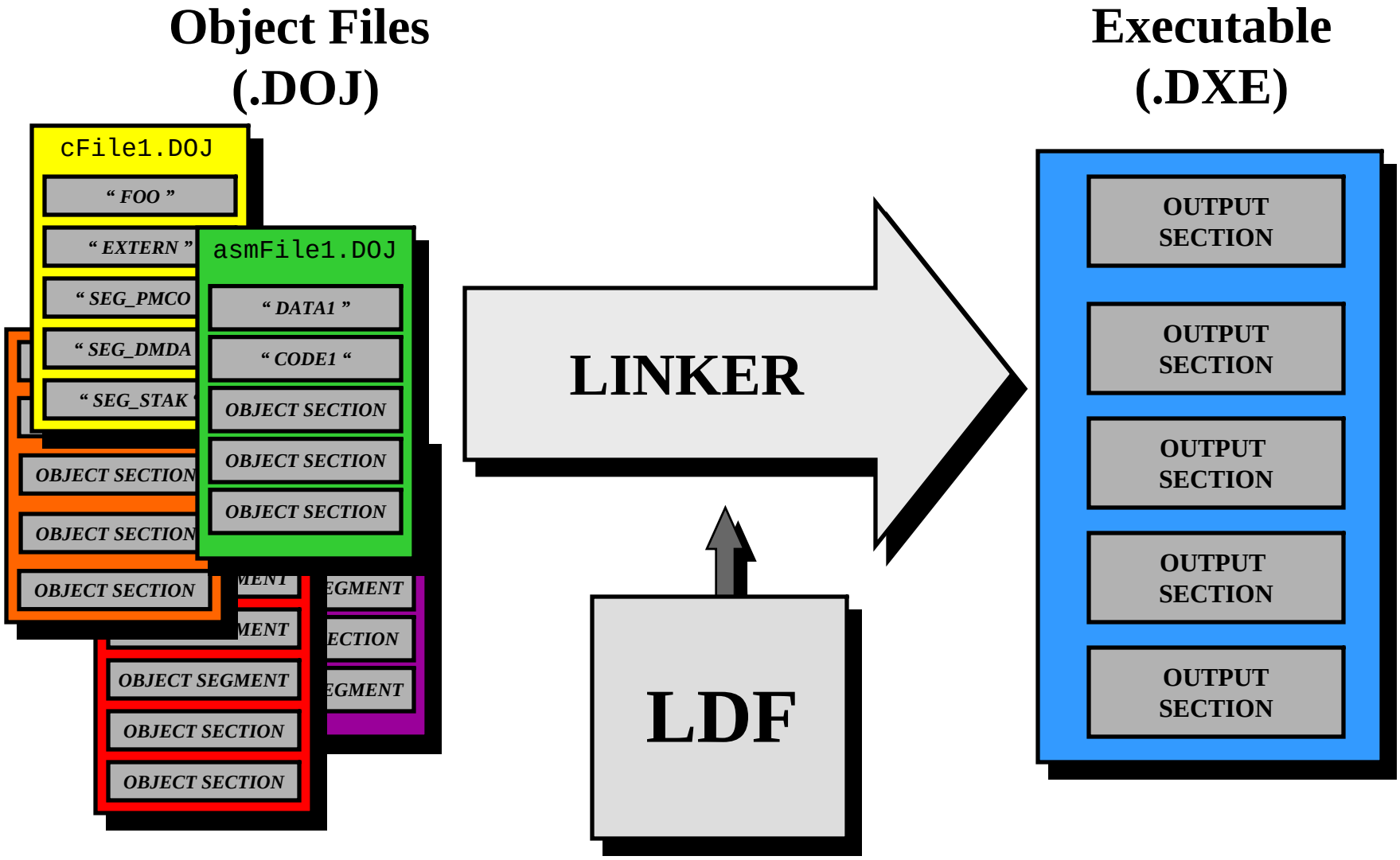
Software Development Flow
Step 2 - Linking



—Analog Devices Confidential Information—

Linker Description File

Step 2 - Linking



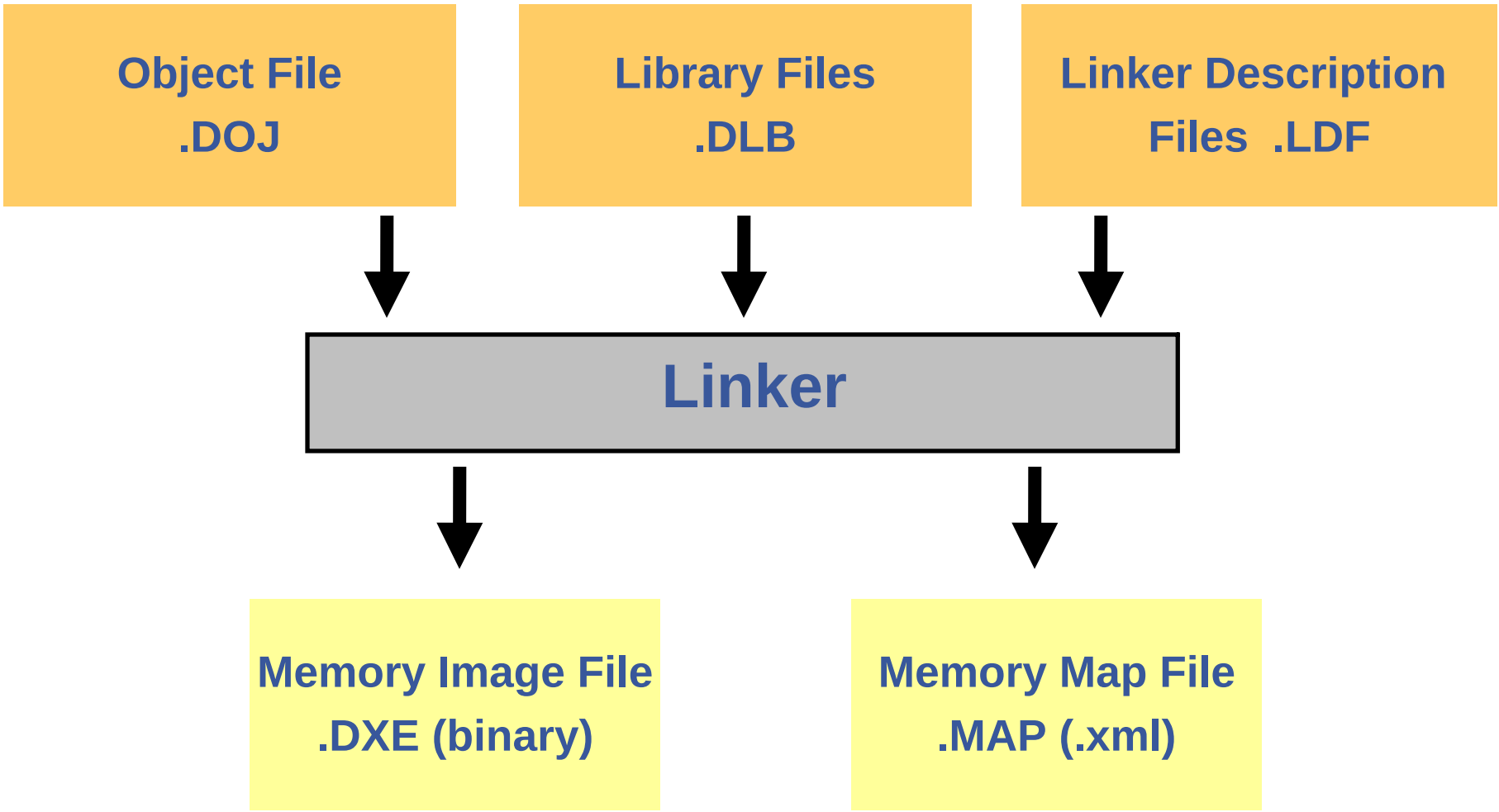
—Analog Devices Confidential Information—

Linker

- **Generates a Complete Executable DSP Program (.dxe)**
- **Resolves All External References**
- **Assigns Addresses to re-locatable Code and Data Spaces**
- **Generates Optional Memory Map**
- **Output in ELF format**
  - **Used by downstream tools such as Loader, Simulator, and Emulator**
- **Controlled by linker commands contained in a linker description file (LDF)**
  - **An LDF is required for each project**
  - **Typically modify a default one to suit target application**

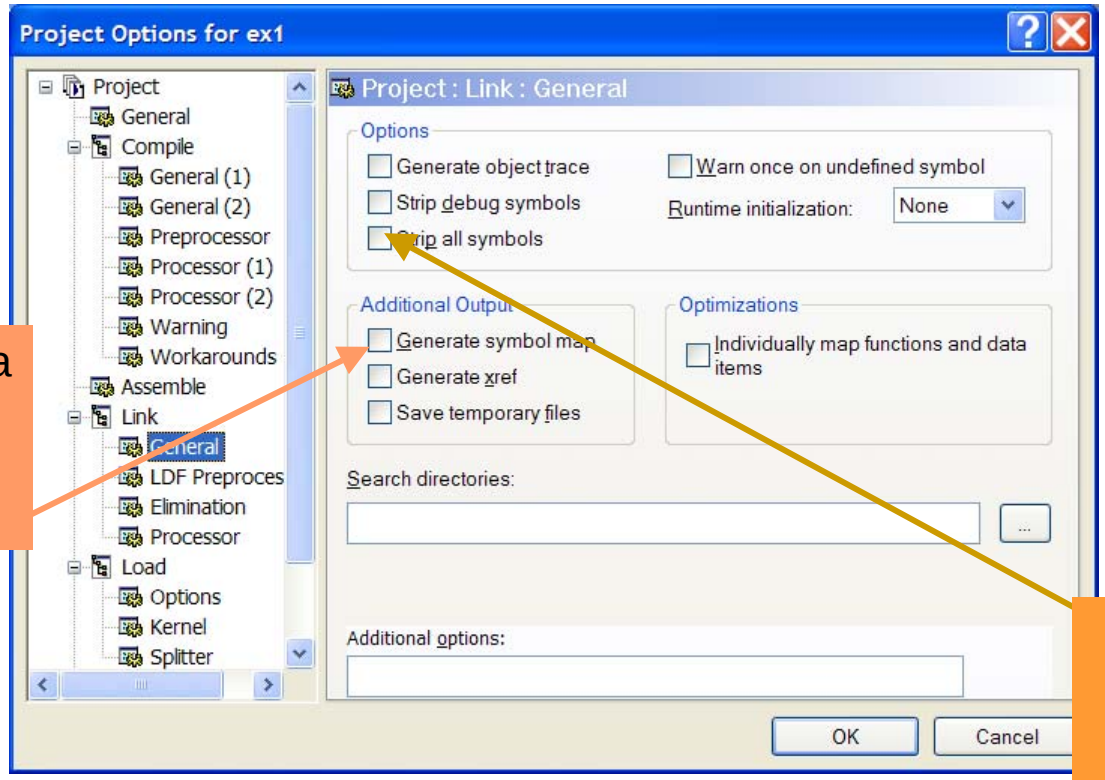
ANALOG DEVICES

Linker



—Analog Devices Confidential Information—

Linker Property Page



If chosen, a .map file will be created

All symbol names will be removed, if chosen

The Linker Description File (LDF)

- **The link process is controlled by a linker command language**
- **The LDF provides a complete specification of mapping between the linker's input files and its output.**
- **It controls**
  - input files
  - output file
  - target memory configuration
- **Preprocessor Support**

LDF consists of three primary parts

- **Global Commands**
  - Defines architecture or processor
  - Directory search paths
  - Libraries and object files to include
- **Memory Description**
  - Defines memory segments
- **Link Project Commands**
  - Mapping of input sections to memory segments
  - Output file name
  - Link against object file list

Example LDF

Global Commands & Memory Description

```
ARCHITECTURE (ADSP-BF533)
SEARCH_DIR ($ADI_DSP\Blackfin\lib)
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY
{
    seg_data_a      { TYPE(RAM) START(0xFF800000)  END(0xFF803FFF) WIDTH(8) }
    seg_data_b      { TYPE(RAM) START(0xFF900000)  END(0xFF903FFF) WIDTH(8) }
    seg_data_scr    { TYPE(RAM) START(0xFFB00000) END(0xFFB00FFF) WIDTH(8) }
    seg_prog        { TYPE(RAM) START(0xFFA00000) END(0xFFA03FFF) WIDTH(8) }
}
```

Global Commands

Segment name

Start address

End address

Memory width

ANALOG DEVICES

Example LDF (con't)

Link Commands

```
PROCESSOR p0
{
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS

{
        sec_data_a
        { INPUT_SECTIONS( $OBJECTS(data_a) ) } > seg_data_a
        sec_data_b
        { INPUT_SECTIONS( $OBJECTS(data_b) ) } > seg_data_b
        sec_data_scr
        { INPUT_SECTIONS( $OBJECTS(data_scr) ) } > seg_data_scr
        sec_prog
        { INPUT_SECTIONS( $OBJECTS(prog) ) } >seg_prog
    }
}
```

OBJECT SECTIONS from assembly files

MEMORY SEGMENTS Declared in the LDF

DXE SECTION NAMES Used in .map file

ANALOG DEVICES

Linker Description File for C/C++ Programming

- **Memory Description**
  - Define Memory Segments
  - Map Input Sections (Names Produced by Compiler) to Memory Segments
- **Run Time Stack Supported**
  - Stack Used for Branching, Local Variables, Arguments
  - LDF Defines Stack Size and Location
- **Run Time Heap Supported**
  - Used For Memory Management Protocols (malloc, free, etc)
  - LDF Defines Heap Size, Location, and Name (For Multiple Heap Support)

—Analog Devices Confidential Information—

Compiler-Generated Memory Section Names

- **Compiler uses default section names that are mapped appropriately by the linker (through the LDF)**
  - **program - contains all program instructions**
  - **data1 - contains all global and "static" data**
  - **constdata - contains all data declared as "const"**
  - **ctor - C++ constructor initializations**
  - **cplb_code – code CPLB config tables**
  - **cplb_data – data CPLB config tables**

Memory Descriptions

- **Define Memory Segments In LDF For:**
  - **Code, Data, Stack*, Heap(s)**
- **Map Input Sections to Memory Segments**
  (BF533 Default LDF Segment Names Used)

Segment Name	Use
MEM_L1_CODE	code storage
MEM_L1_CODE_CACHE	code storage, if not cache
MEM_L1_DATA_A	used for default compiler data sections
MEM_L1_DATA_A_CACHE	If not used as cache, it becomes heap space
MEM_L1_DATA_B	used for default compiler data sections
MEM_L1_DATA_B_CACHE	If not used as cache, it is used for data
MEM_L1_DATA_B_STACK	dedicated stack space
MEM_L1_SCRATCH	Dedicated 4 Kbyte Data Scratchpad
MEM_SDRAM0_HEAP	If L1 Data A used as cache, heap is external
MEM_SDRAM0	external SDRAM bank
MEM_ASYNCx	(x=0,1,2,3) 1MB Async Banks

LDF and the Stack

- **C/C++ Runtime Environment Depends Upon the Initialization of FP and SP**
- **Variables Initialized by Constants Defined in the LDF**
  - ldf_stack_space
  - ldf_stack_end
- **Variables Used to Initialize FP and SP are Declared and Initialized in the Assembly File basiccrt.s**

LDF Stack Setup (C/C++ Compiler Only)

- **Linker Calculates LDF Stack-Initializing Constants from the Stack Memory Segment Description**

```
stack
    {
        ldf_stack_space = .;
        ldf_stack_end = ldf_stack_space +
MEMORY_SIZEOF(MEM_L1_DATA_B_STACK);
    } >MEM_L1_DATA_B_STACK
```

LDF and the Heap

- **Four Library Functions Can Be Used to Allocate or Free Memory to/from the Heap**
  - **malloc, calloc, realloc, free**
- **Other C Library Functions Implicitly Use these Four Functions and ALSO Require the Heap**
  - **memmove, memcopy, etc.**
- **Initialized by Constants Defined in the LDF**
  - **ldf_heap_space**
  - **ldf_heap_length**
  - **ldf_heap_end**
- **Multiple Heaps are Possible**
  - **Can be defined at Link Time or at Run Time (see compiler manual)**

ANALOG DEVICES

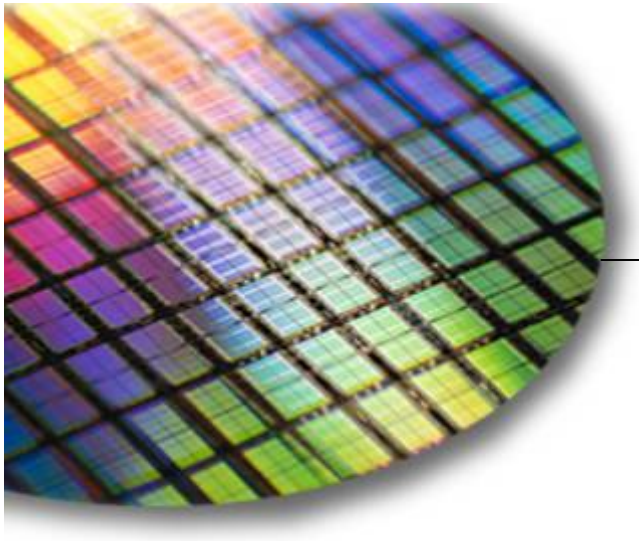
LDF Heap Setup

(C Compiler Only)

- **Output Section 'heap' Calculates LDF Heap Initializers from Heap Memory Segment Description**

```
#ifdef USE_CACHE /* { */
        heap
        {
            // Allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_SDRAM0_HEAP) - 1;
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } >MEM_SDRAM0_HEAP
#else
        heap
        {
            // Allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_L1_DATA_A_CACHE) - 1;
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } >MEM_L1_DATA_A_CACHE
#endif /* USE_CACHE } */
```


ANALOG DEVICES



The World Leader in High Performance Signal Processing Solutions

Expert Linker

Using the LDF Wizard

ADI Confidential Information – Not for external distribution

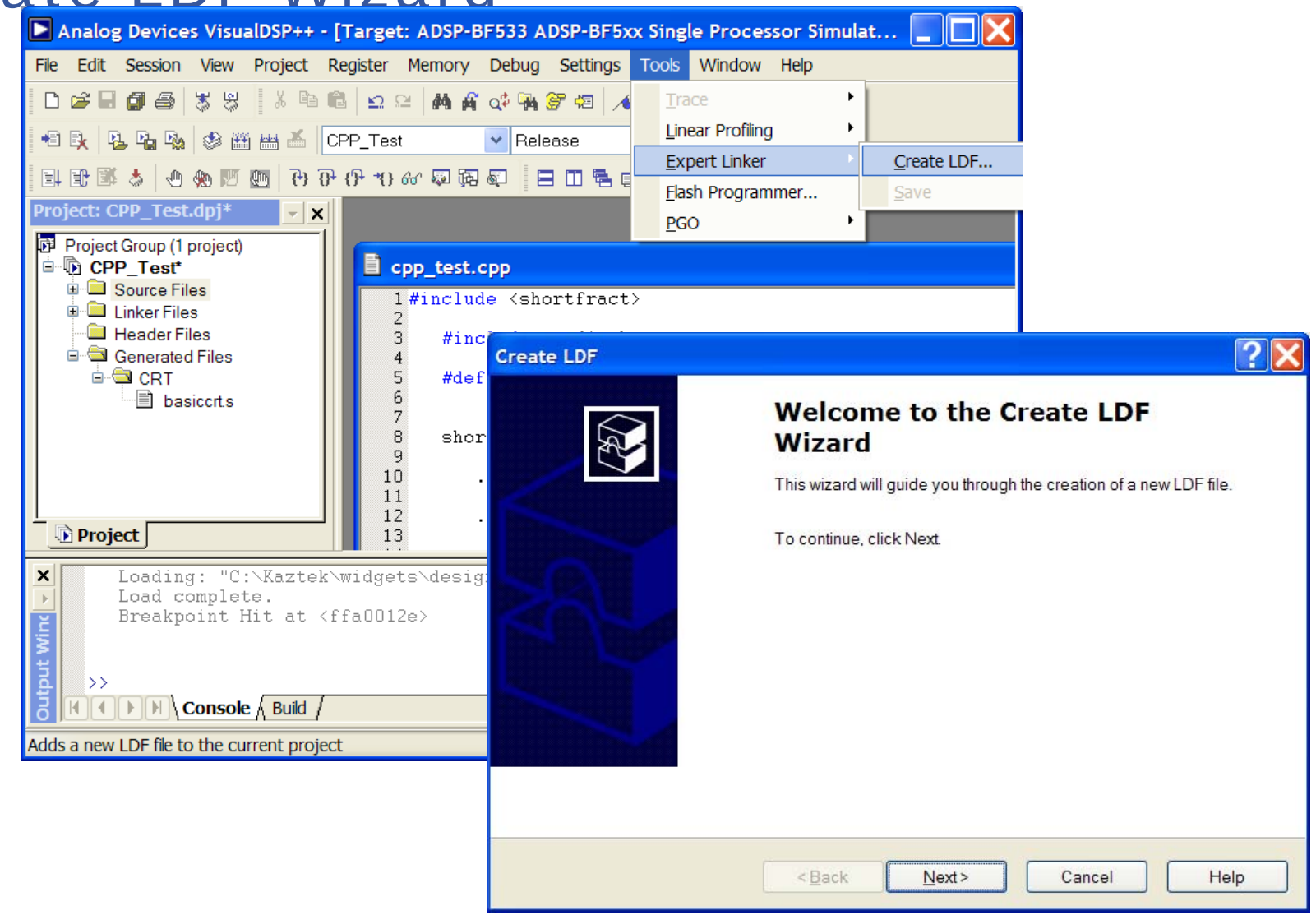
ANALOG DEVICES

Expert Linker Features

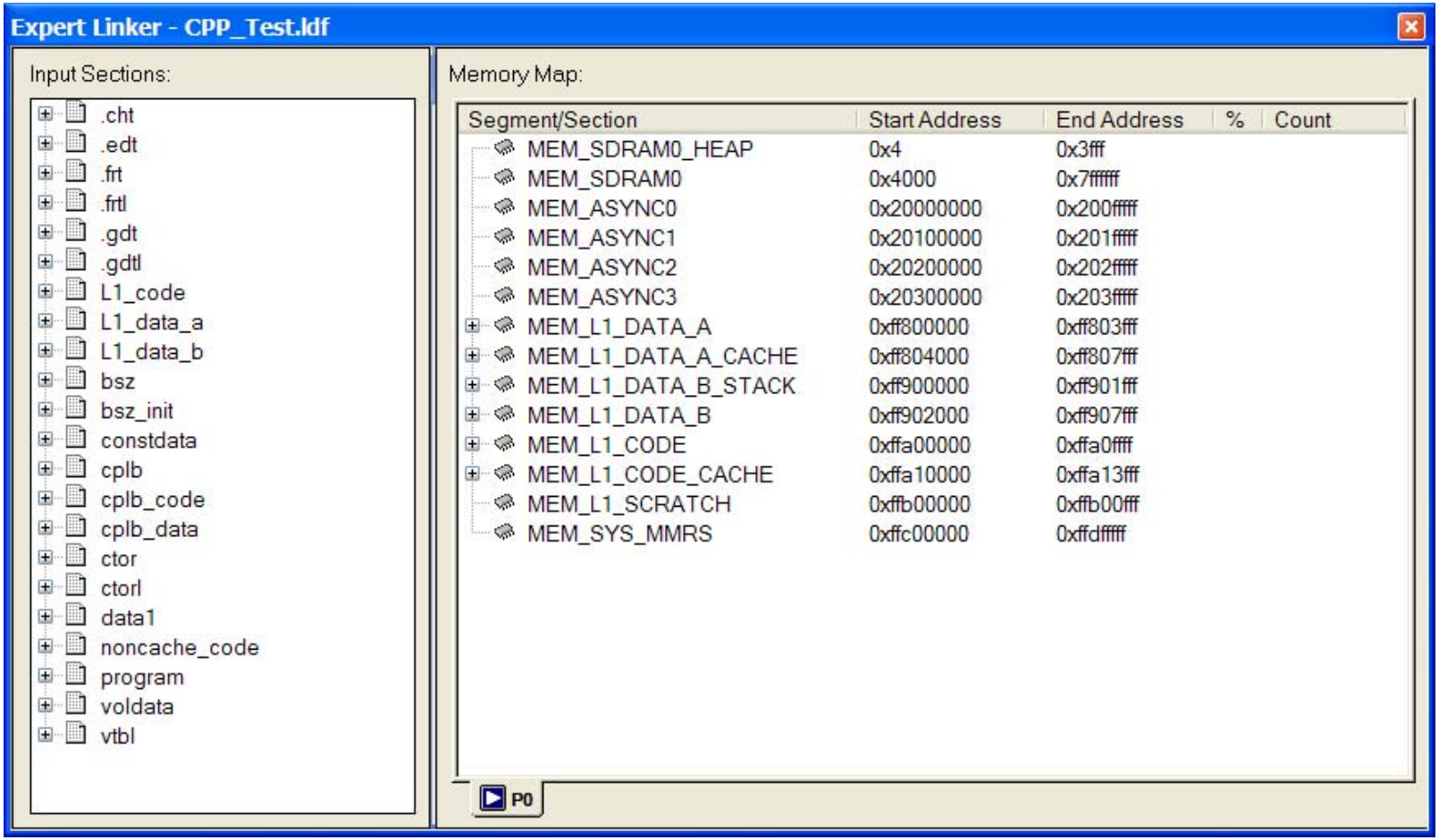
Expert Linker is a Graphical tools that can:

- **Use wizards to create LDF files**
- **Define a DSP's target memory map**
- **Drag and Drop object sections into the memory map**
- **Graphically highlights code elimination of unused objects**
- **Profile object sections in memory**

Create LDF Wizard



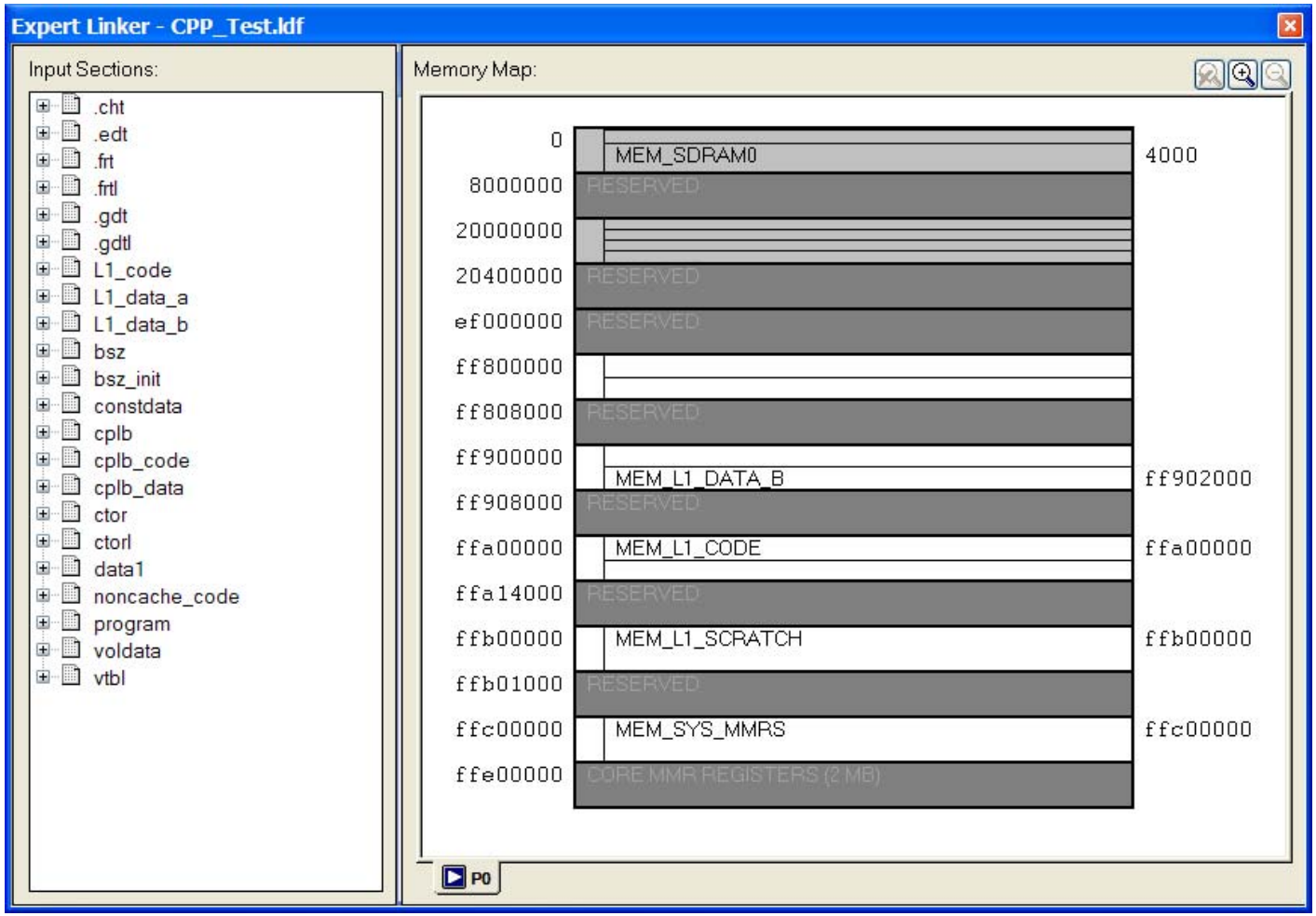
LDF Result



This is a memory map view of the generated .ldf file. In this mode, each section's start and end address are shown in a list format.

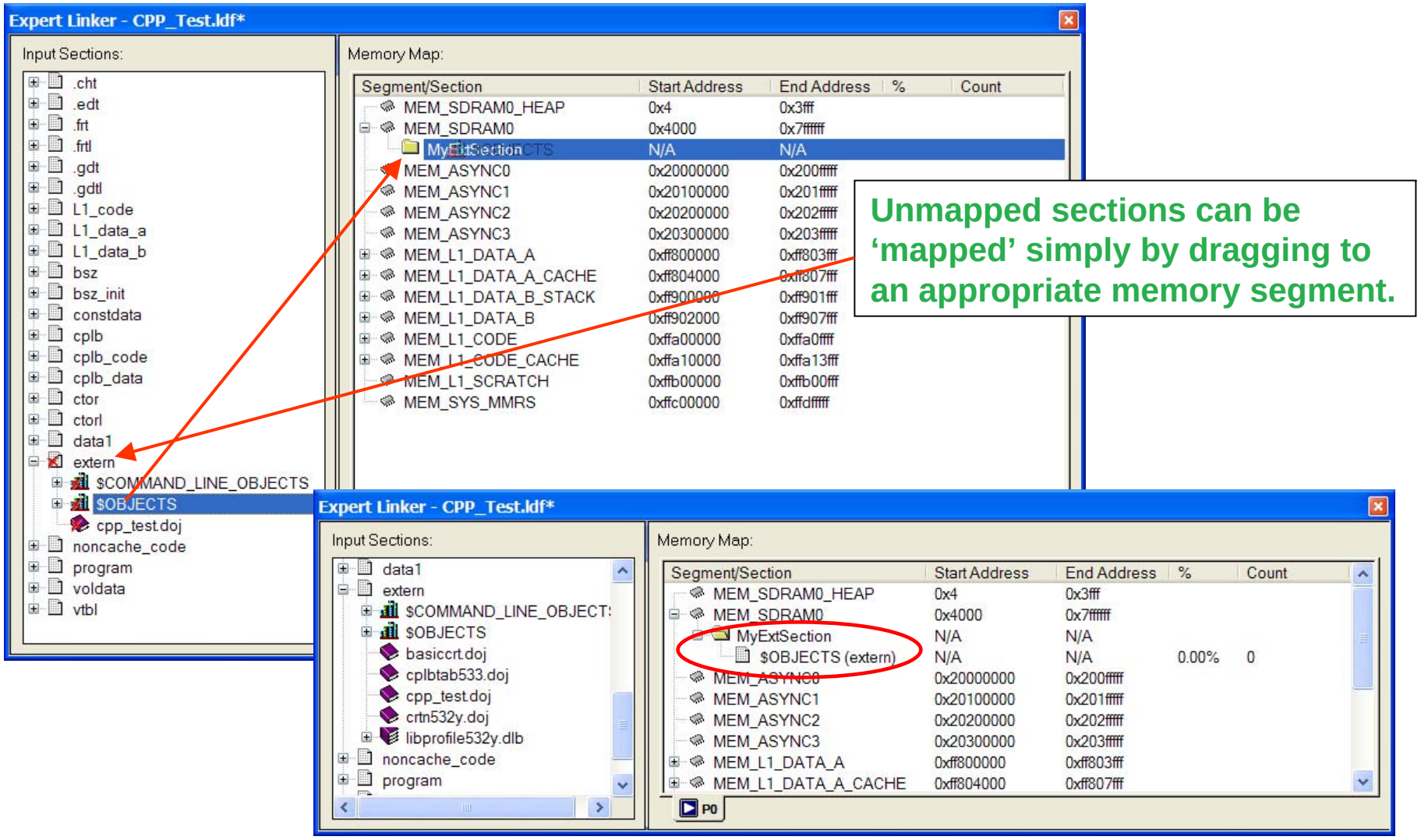
—Analog Devices Confidential Information—

LDF Result (cont'd)

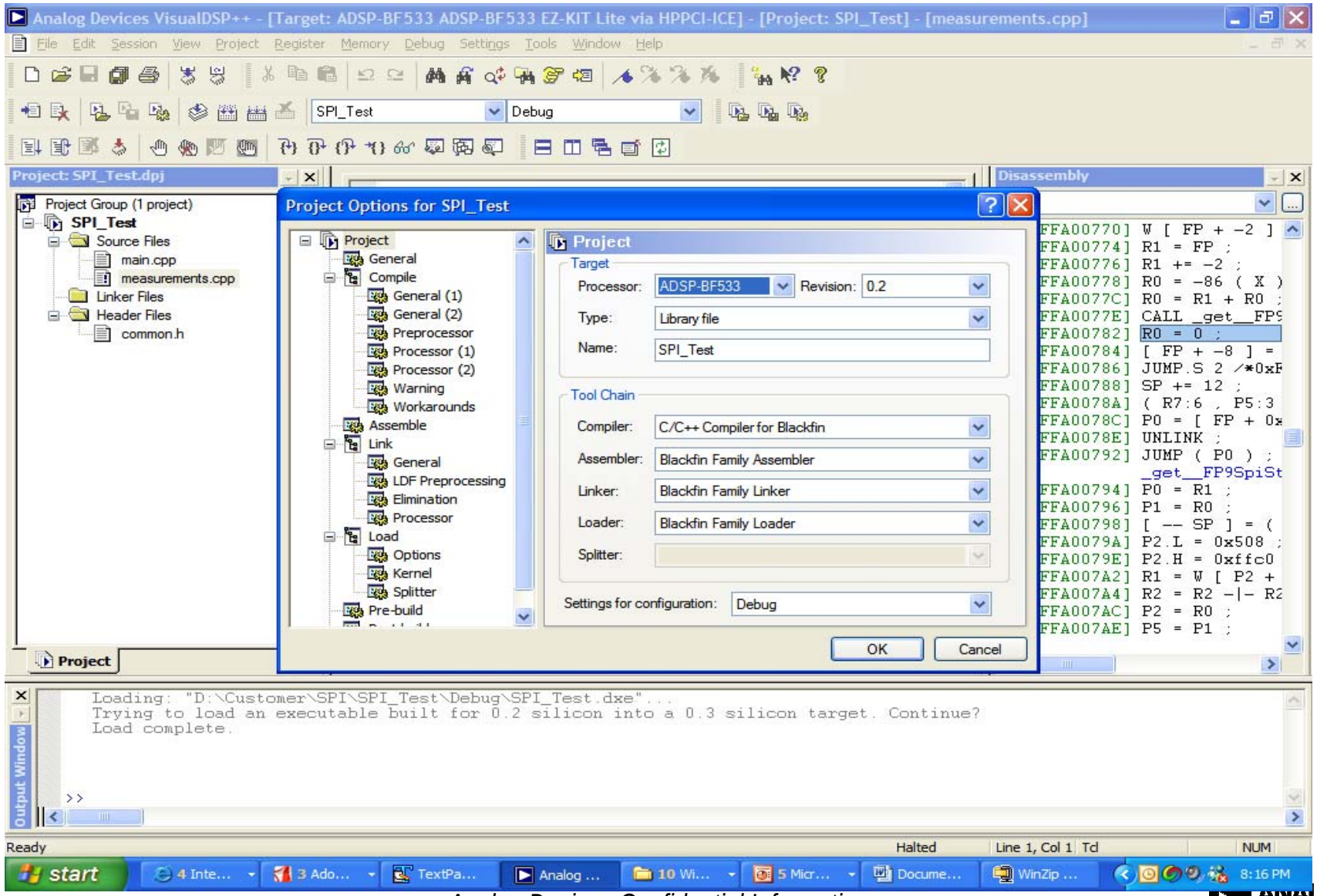


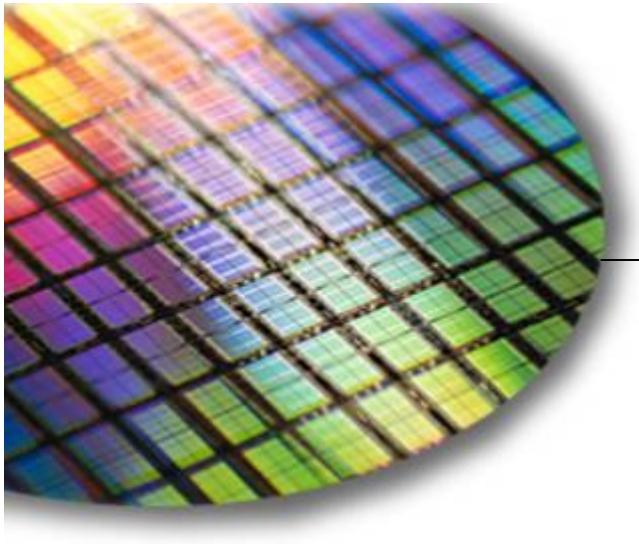
This is a graphical view of the memory map. Double click on the section to zoom in.

Control Mapping of Sections



How to create Library Functions





The World Leader in High Performance Signal Processing Solutions

Section 11

Direct Memory Access (DMA)

ADI Confidential Information – Not for external distribution

ANALOG DEVICES

BF533 DMA Overview

- **The ADSP-BF533 DMA controller allows data transfer operations without processor intervention**
  - **Core sets up registers or descriptors**
  - **Core responds to interrupts when data is available**
- **Types of data transfers**

Internal or External Memory	**↔**	**Internal or External Memory**
Internal or External Memory	**↔**	**Serial Peripheral Interface (SPI)**
Internal or External Memory	**↔**	**Serial Port**
Internal or External Memory	**↔**	**UART Port**
Internal or External Memory	**↔**	**Parallel Port Interface (PPI)**

Overview (cont.)

- **The ADSP-BF533 system includes 6 DMA-capable peripherals, including the Memory DMA controller (MemDMA) with 12 DMA channels and bus masters that support these devices:**

SPORT0 RCV DMA Channel
SPORT1 RCV DMA Channel
SPORT0 XMT DMA Channel
SPORT1 XMT DMA Channel
SPI DMA Channel
Streams

UART RCV Channel
UART XMT Channel
PPI DMA Channel
4 Memory DMA Channels

- Equates to 2 DMA

BF533 DMA Buses

- **The DMA Access Bus (DAB) provides a means for DMA channels to be accessed by the peripherals.**
- **The DMA External Bus (DEB) provides a means for DMA channels to gain access to off-chip memory.**
  - **The core processor has priority over the DEB on the External Port Bus (EPB) for off-chip memory.**
- **The DMA Core Bus (DCB) provides a means for DMA channels to gain access to on-chip memory.**
  - **The DCB has priority over the core processor on arbitration into L1 memory configured as SRAM.**

BF533 DMA Priority

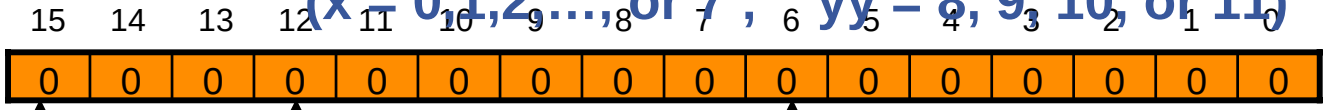
The ADSP-BF533 processor uses the following priority arbitration policy on the DAB.

DMA Channel	Default Peripheral Mapping	Comments
0 – highest	PPI	Re-assignable
1	SPORT0 RX	Re-assignable
2	SPORT0 TX	Re-assignable
3	SPORT1 RX	Re-assignable
4	SPORT1 TX	Re-assignable
5	SPI	Re-assignable
6	UART RX	Re-assignable
7	UART TX	Re-assignable
8	Memory DMA Stream 0 TX (destination)	Fixed
9	Memory DMA Stream 0 RX (source)	Fixed
10	Memory DMA Stream 1 TX (destination)	Fixed
11 - lowest	Memory DMA Stream 1 RX (source)	Fixed

Peripheral Map Register

DMAx_PERIPHERAL_MAP / MDMA_yy_ PERIPHERAL_MAP

(x = 0,1,2…,or 7 ; yy = 8, 9, 10, or 11)



Reset = 0x0000

CTYPE (DMA Channel Type) –
RO 0 – Peripheral DMA
1 – Memory DMA

PMAP[3:0] (Peripheral Mapped to this Channel)

0000 – PPI
0001 – SPORT0 RX
0010 – SPORT0 TX
0011 – SPORT1 RX
0100 – SPORT1 TX
0101 – SPI
0110 – UART RX
0111 – UART TX

- **The Peripheral Map Register allows the user to map a peripheral to a specific channel thus programming the priority of each peripheral.**

DMA Initialization

- **To initiate a DMA transfer, certain parameters need to be defined before the DMA engine can start a DMA sequence. These parameters are:**
  - **Configuration**
    - describes certain characteristics of the DMA transfer such as data size, transfer direction, etc..
  - **Start Address**
    - Specifies the address where the DMA transfer will start from.
  - **Count**
    - Specifies the number of elements the DMA Engine will transfer.
  - **Modify**
    - Specifies the address increment after every element transfer

—Analog Devices Confidential Information—

ANALOG DEVICES

DMA Schemes

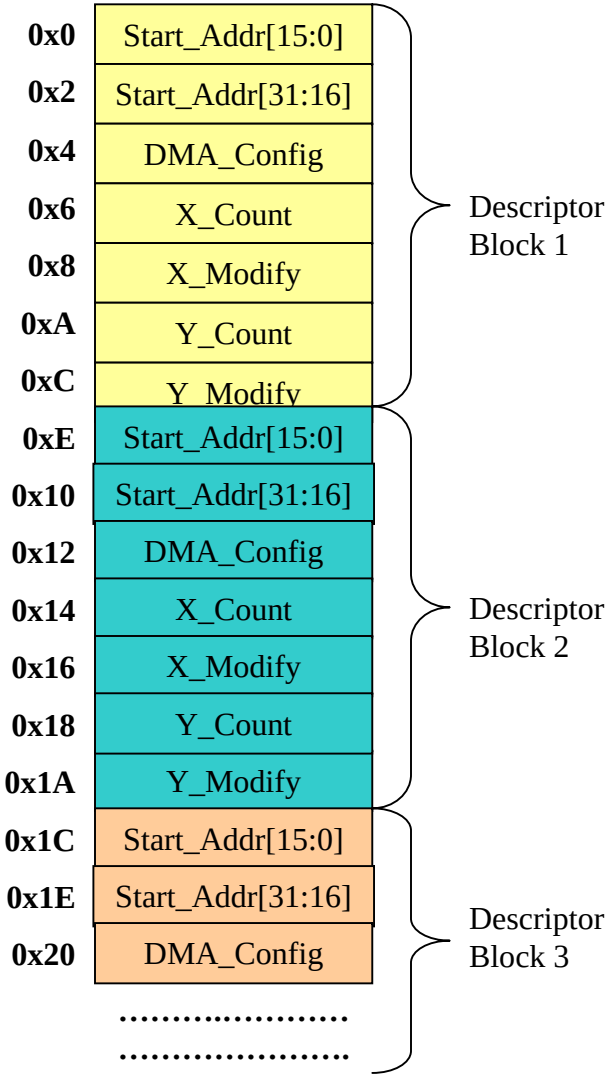
Two Types of DMA transfers available on the ADSP-BF533/BF561

- **Descriptor-based DMA transfers**
  - **Requires a set of parameters stored within memory to initiate a DMA sequence. These parameters are transferred to DMA control registers upon a start of a DMA transfer.**
  - **Supports chaining of multiple DMA transfers.**
- **Register-based DMA transfers**
  - **Allows the user to program the DMA control registers directly to define and initiate a DMA sequence.**
  - **Upon DMA completion, depending on certain bits with the Configuration Register:**
    - Control registers are automatically updated with their original setup values (Autobuffer Mode).
    - Or the DMA Channel gracefully shuts off (Stop Mode).

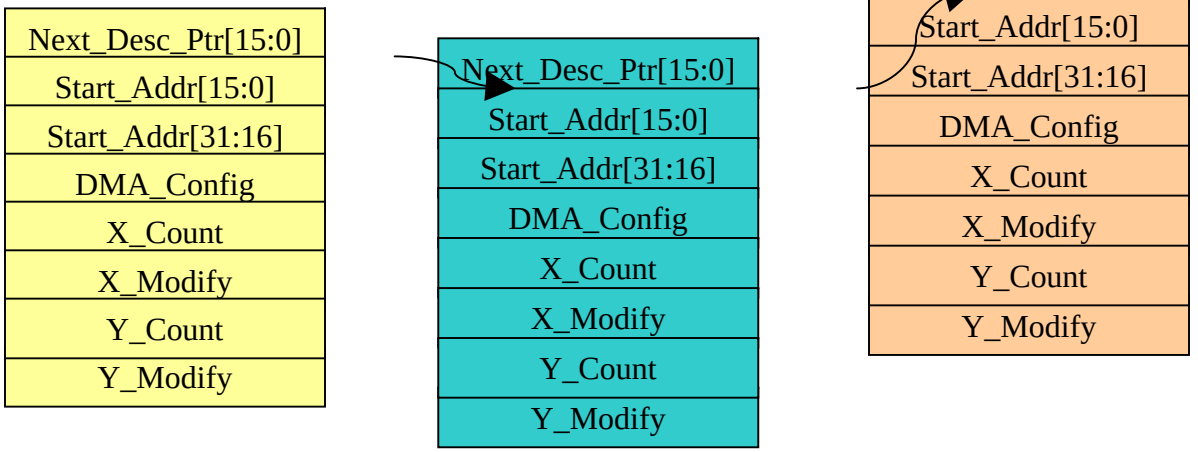
ANALOG DEVICES

Descriptor Blocks

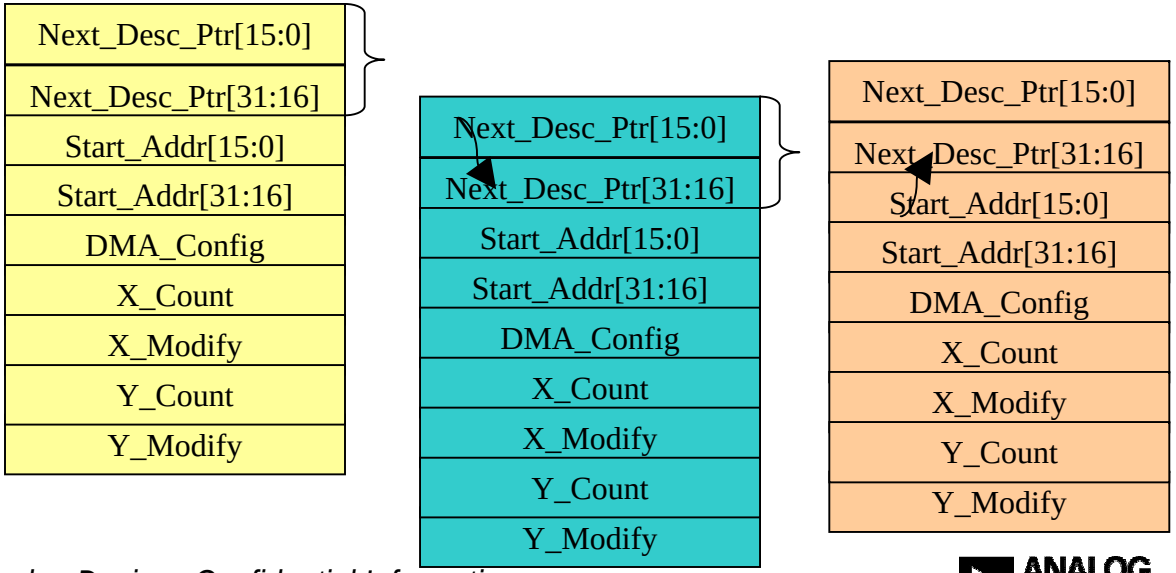
Descriptor Array Mode



Descriptor List (Small Model) Mode



Descriptor List (Large Model) Mode



—Analog Devices Confidential Information—

Transfer Modes

- **The Transfer Mode is controlled by 3 bits called the FLOW[2:0] bits within the DMA Configuration Register.**
  - **Stop Mode (FLOW = 0x0).**
    - When the current DMA transfer completes, the DMA channel stops automatically, after signaling an interrupt if enabled.
  - **Autobuffer Mode (FLOW = 0x1).**
    - DMA is performed in a continuous circular-buffer fashion based on user-programmed DMAx MMR settings. On completion of the DMA transfer, the Parameter registers are reloaded into the Current registers, and DMA resumes immediately with zero overhead. Autobuffer mode is stopped by a user write of 0 to the DMA enable bit in the DMAx_DMA_Config Register.
  - **Descriptor Array Mode (FLOW = 0x4).**
    - In this mode, the Descriptor Block does not include the NEXT_DESC_PTR parameter. Descriptor Blocks are placed one after the other within memory like an array.
  - **Descriptor List (Small Model) Mode (FLOW = 0x6).**
    - In this mode, the Descriptor Block does not include the upper 16 bits of the NEXT_DESC_PTR parameter. The upper 16 bits are taken from the upper 16 bits of the NEXT_DESC_PTR register, thus confining all descriptors to a specific 64K page in memory.
  - **Descriptor List (Large Model) Mode (FLOW = 0x7).**
    - In this mode, Descriptor Block includes all 32 bits of the NEXT_DESC_PTR parameter, thus allowing maximum flexibility in locating descriptors in memory.

Descriptor Block Structures

- **Depending on the Descriptor Mode used, the following lists the order of the Descriptor Block Parameters stored within memory:**

Descriptor Offset	Descriptor Array Mode (FLOW = 0x4)	Small Descriptor List Mode (FLOW = 0x6)	Large Descriptor List Mode (FLOW = 0x7)
0x0	START_ADDR[15:0]	NEXT_DESC_PTR[15:0]	NEXT_DESC_PTR[15:0]
0x2	START_ADDR[31:16]	START_ADDR[15:0]	NEXT_DESC_PTR[31:16]
0x4	DMA_CONFIG	START_ADDR[31:16]	START_ADDR[15:0]
0x6	X_COUNT	DMA_CONFIG	START_ADDR[31:16]
0x8	X_MODIFY	X_COUNT	DMA_CONFIG
0xA	Y_COUNT	X_MODIFY	X_COUNT
0xC	Y_MODIFY	Y_COUNT	X_MODIFY
0xE		Y_MODIFY	Y_COUNT
0x10			Y_MODIFY

NOTE: Not all of the Parameters need to be initialized within the Descriptor Block depending on the NDSIZE value within the DMA Configuration Register. The NDSIZE value is the number of Parameters that the DMA engine will fetch for the next Descriptor Block

Analog Devices Confidential Information

DMA Register Setup

To start DMA operation, some or all of the DMA Parameter Registers must first be initialized depending on the 'Next Descriptor Size'(NDSIZE) and 'FLOW' bits in the DMA Configuration Register. After Initialization, DMA operation begins by writing a 1 to the DMA Enable bit in the DMA Configuration Register.

1) **FLOW = 0x0 (Stop Mode), NDSIZE = 0x0:**

Initialize all of the following:

- START_ADDR
- X_COUNT
- X_MODIFY
- Y_COUNT (if 2D DMA)
- Y_MODIFY (if 2D DMA)
- DMA_CONFIG

2) **FLOW = 0x1 (Autobuffer Mode), NDSIZE = 0x0:**

Initialize all of the following:

- START_ADDR
- X_COUNT
- X_MODIFY
- Y_COUNT (if 2D DMA)
- Y_MODIFY (if 2D DMA)
- DMA_CONFIG

3) **FLOW = 0x4 (Descriptor Array Mode), NDSIZE = 0x0 – 0x7:**

Initialize at least:

- CURR_DESC_PTR[31:16]
- CURR_DESC_PTR[15:0]

4) **FLOW = 0x6 (Small Descriptor List Mode), NDSIZE = 0x0 – 0x8:**

Initialize at least:

- NEXT_DESC_PTR[31:16]
- NEXT_DESC_PTR[15:0]

5) **FLOW = 0x7 (Large Descriptor List Mode), NDSIZE = 0x0 – 0x9:**

Initialize at least:

- NEXT_DESC_PTR[31:16]
- NEXT_DESC_PTR[15:0]

How to Stop DMA Transfers

- **FLOW = 0x0 (Stop Mode):**
  - **DMA stops automatically after the DMA transfer is complete.**
- **FLOW = 0x1 (Autobuffer Mode):**
  - **Write a 0 to the DMA Enable bit in the DMA Configuration Register. A write of 0x0 to the entire register will always terminate DMA gracefully (without DMA Abort).**
- **FLOW = 0x4, 0x6, 0x7 (Array / List Mode):**
  - **Set the final DMA_CONFIG Register with FLOW = 0x0 setting to gracefully stop the DMA channel. If the DMA_CONFIG Parameter is not included within the Descriptor Block, use the FLOW = 0x1 method above to end the DMA.**

Memory DMA (MemDMA)

- **Allows memory-to-memory DMA transfers between the various ADSP-BF533 memory spaces**
- **A single MemDMA transfer requires a pair of DMA channels:**
  - **One to specify the Source block of memory**
  - **One to specify the Destination block of memory**
- **ADSP-BF533 consists of four MemDMA channels which allows setup for 2 memory-to-memory DMA transfers at the same time**
  - **Two Source DMA Channel – used to read from memory**
  - **Two Destination DMA Channel – used to write to memory**
- **Both the Source and Destination DMA Channels share a 8-entry, 16-bit FIFO (32-bit FIFOs on the BF561)**
  - **Source DMA Channel fills the FIFO**
  - **Destination DMA Channel empties the FIFO**

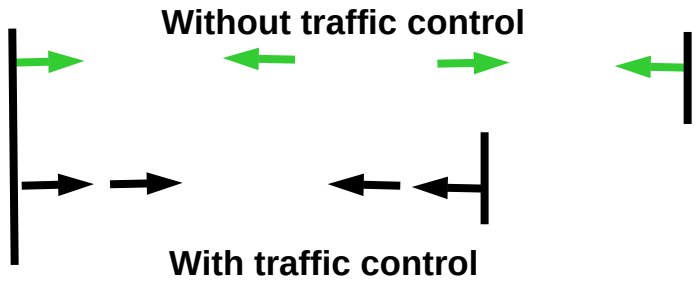
Memory DMA (MemDMA)

- **Each DMA transfer sequence requires two sets of Descriptor Blocks within memory**
  - **One for the Source DMA Channel**
  - **One for the Destination DMA Channel**
  - **Both sets of Descriptor Blocks must be configured to have the same transfer count and data size but they can have different modify values.**
  - **The DMA Configuration Register for the source channel must be written before the DMA Configuration Register for the destination channel. When the destination DMA Configuration Register is written, MemDMA operations starts after a latency of 3 SCLK cycles**
- **It is preferable to activate interrupts on only one channel**
  - **Eliminates ambiguity when trying to identify the channel (either source or destination) that requested the interrupt**

Prioritization and Traffic Control

- **Traffic can be independently controlled for each of the three buses (DAB, DCB, and DEB) with simple counters**
  - **alternation of transfers between MDMA streams can also be controlled**
- **Using the traffic control features, the DMA system preferentially grants data transfers on the DAB or memory buses (DCB and DEB), which are going in the same read/write direction as the previous transfer, until either the traffic control counter times out, or until traffic stops or changes direction on its own.**
- **When a count field in TC_CNT expires, it is automatically reloaded with the appropriate value programmed in TC_PER (ie period value).**
- **When a DAB, DEB, or DCB counter decrements from 1 to 0, the opposite-direction DAB, DCB, or DEB access is preferred,**
  - **This may result in a direction change.**
- **When the MDMA counter decrements from 1 to 0, the next available MDMA stream is selected.**
  - **If the MDMA period is set to 0, then MDMA is scheduled by fixed priority.**
  - **If the MDMA period is set between 1<=p<=31, the two MDMA streams are granted bus access in alternate bursts of up to p data transfers**

Traffic Control (cont'd)



Arrows represent transfers in and out of SDRAM

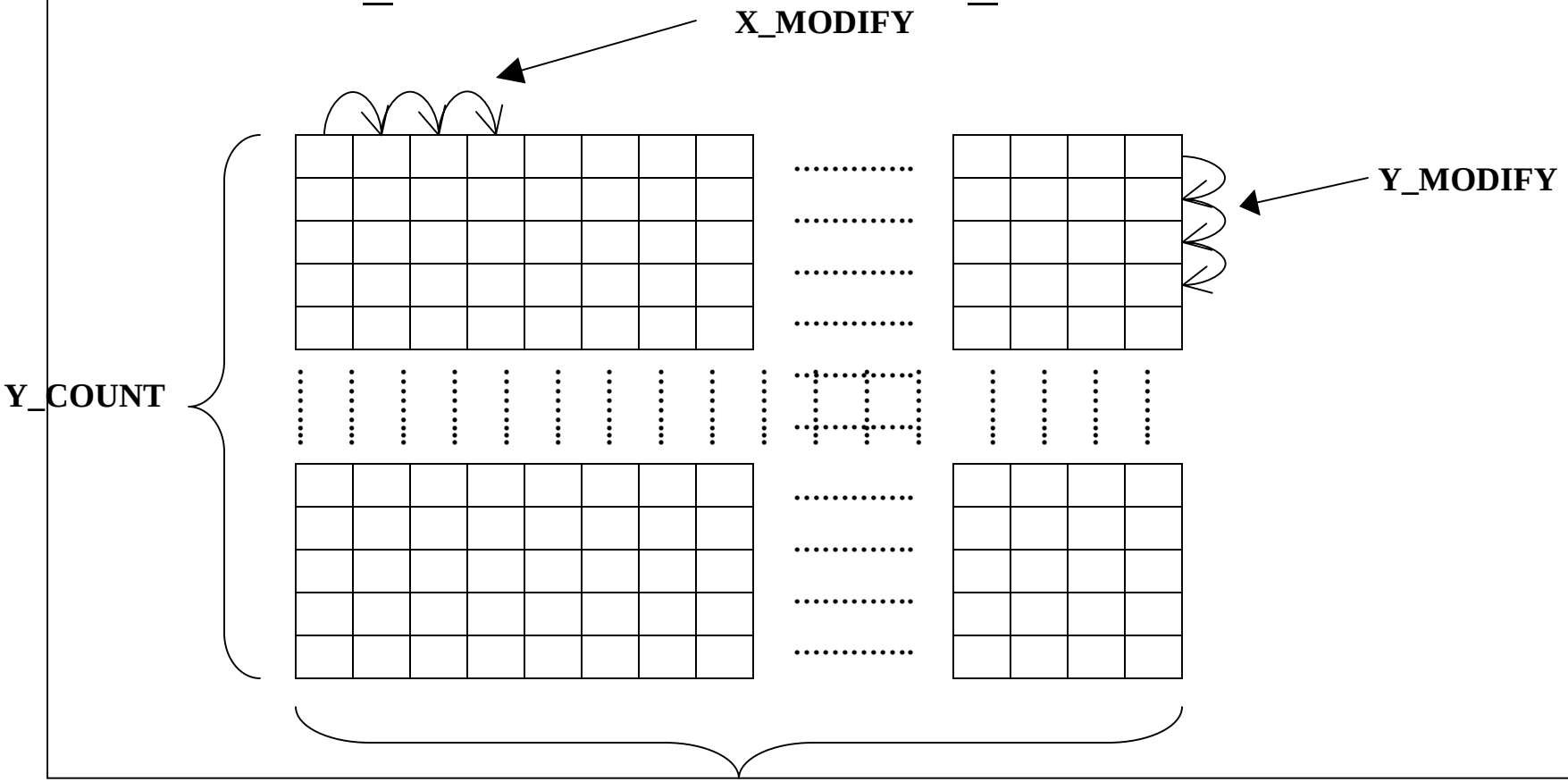
Important Register: Allows the definition of transfer sizes in a given direction on DMA busses

Max values usually yield best performance but it is application dependent

2 Reads and 2 writes are more efficient with traffic control

Two-Dimensional DMA (2D DMA)

Supports arbitrary row and column sizes up to 64K x 64K elements. X_Count = row size and Y_Count = column size.



X_COUNT – must be 2 or greater

—Analog Devices Confidential Information—

Two-Dimensional DMA (2D DMA)

- **X_Modify**
  - **is the byte-address increment applied after each transfer that decrements Curr_X_Count.**
  - **is not applied when the inner loop (row) count is ended by decrementing Curr_X_Count from 1 to 0.**
- **Y_Modify**
  - **is the byte-address increment applied after each decrement of Curr_Y_Count.**
  - **is not applied to the last element in the array on which the outer loop (column) count, Curr_Y_Count, also expires by decrementing from 1 to 0.**
- **After the last transfer completes,**
  - **Curr_Y_Count = 1**
  - **Curr_X_Count = 0**
  - **Curr_Addr is equal to the last item's address plus X_Modify.**
- **In Autobuffer Mode, these registers are reloaded from X_Count, Y_Count, and Start_Addr upon the first data transfer.**

BF533 MMRs for Peripheral DMA

MMR Name	Description
DMA*x*_NEXT_DESC_PTR	Link pointer to next descriptor
DMA*x*_START_ADDR	Start address of DMA buffer
DMA*x*_DMA_CONFIG	DMA configuration register
DMA*x*_X_COUNT	Inner loop count
DMA*x*_X_MODIFY	Inner loop address increment, in bytes
DMA*x*_Y_COUNT	Outer loop count (2D DMA only)
DMA*x*_Y_MODIFY	Outer loop address increment, in bytes
DMA*x*_CURR_DESC_PTR	Current Descriptor Pointer
DMA*x*_CURR_ADD	Current DMA Address
DMA*x*_IRQ_STATUS	Interrupt Status Register contains completion and error interrupt status information
DMA*x*_PERIPHERAL_MAP	Priority mapping register
DMA*x*_CURR_X_COUNT	Current count (1D) or intra-row X count (2D)
DMA*x*_CURR_Y_COUNT	Current row count (2D DMA only)

Parameter Registers

Current Registers

Control / Status Registers

ANALOG DEVICES

BF533 MMRs for Memory DMA

MMR Name (yy = S0, S1, D0, D1)	Description
MDMA_yy_NEXT_DESC_PTR	Link pointer to next descriptor
MDMA_yy_START_ADDR	Start address of DMA buffer
MDMA_yy_DMA_CONFIG	DMA configuration register
MDMA_yy_X_COUNT	Inner loop count
MDMA_yy_X_MODIFY	Inner loop address increment, in bytes
MDMA_yy_Y_COUNT	Outer loop count (2D DMA only)
MDMA_yy_Y_MODIFY	Outer loop address increment, in bytes
MDMA_yy_CURR_DESC_PTR	Current Descriptor Pointer
MDMA_yy_CURR_ADD	Current DMA Address
MDMA_yy_IRQ_STATUS	Interrupt Status Register contains completion and error interrupt status information
MDMA_yy_PERIPHERAL_MAP	Priority mapping register (read only)
MDMA_yy_CURR_X_COUNT	Current count (1D) or intra-row X count (2D)
MDMA_yy_CURR_Y_COUNT	Current row count (2D DMA only)

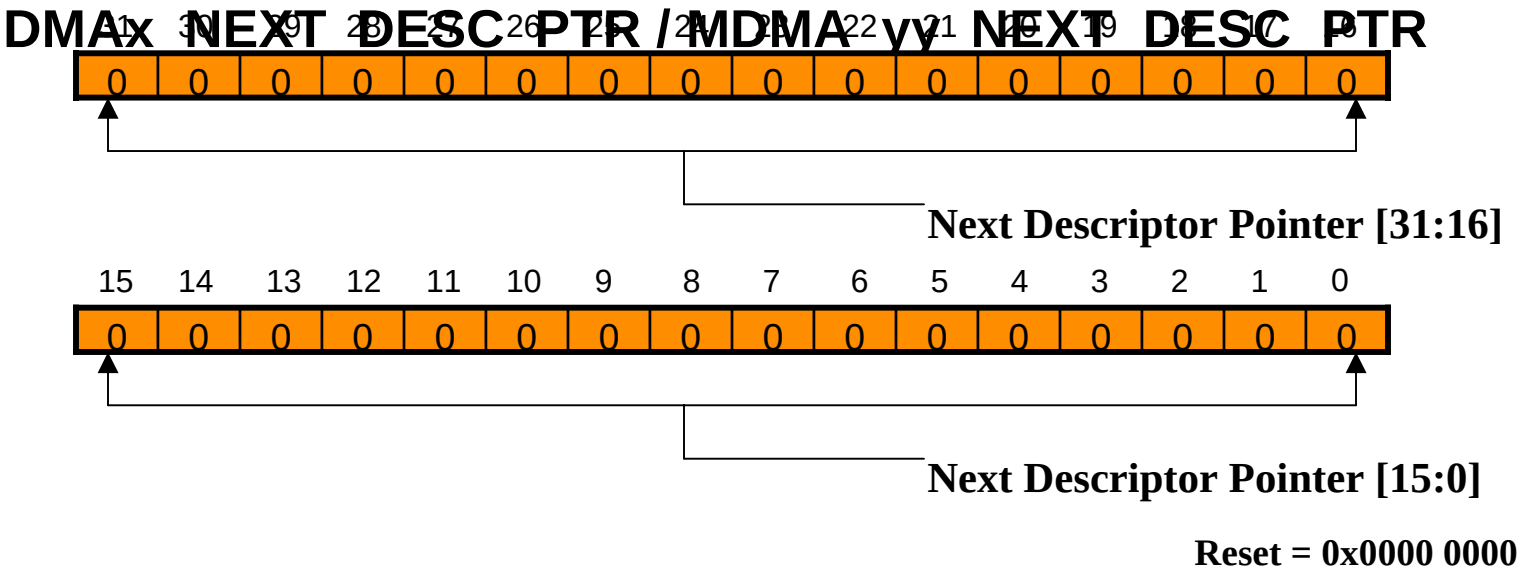
Parameter Registers

Current Registers

Control / Status Registers

ANALOG DEVICES

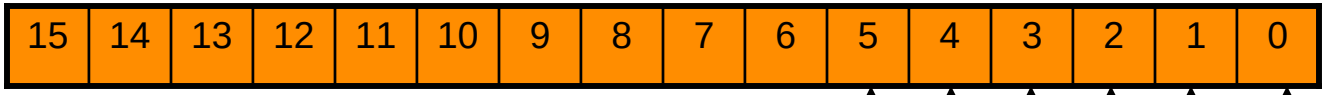
Next Descriptor Pointer Register



- **Specifies the location of the Next Descriptor Block when the current DMA transfer finishes. Used only in Small and Large Descriptor List Modes. Contents of this register are copied into the Curr_Desc_Ptr Register at the start of a descriptor block fetch. Disregarded in Stop, Autobuffer, and Descriptor Array Mode.**

DMA Configuration Register

DMAx_CONFIG / MDMA_yy_CONFIG



DMA Buffer Clear
0 = Retain DMA FIFO data between DMA transfers
1 = Discard DMA FIFO before beginning DMA transfer

DMA Mode
0 = Linear
1 = 2D DMA

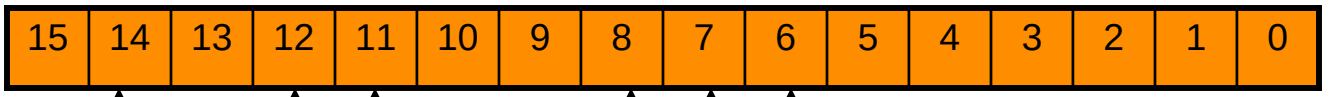
Transfer Word Size
00 = 8-bit transfers
01 = 16-bit transfers
10 = 32-bit transfers
11 = reserved

Transfer Direction
0 = Memory Read
1 = Memory Write
(Bit 1 cannot be modified for some peripherals and MemDMA)

DMA Enable
0 = Disabled
1 = Enabled

DMA Configuration Register (cont.)

DMAx_CONFIG / MDMA_yy_CONFIG



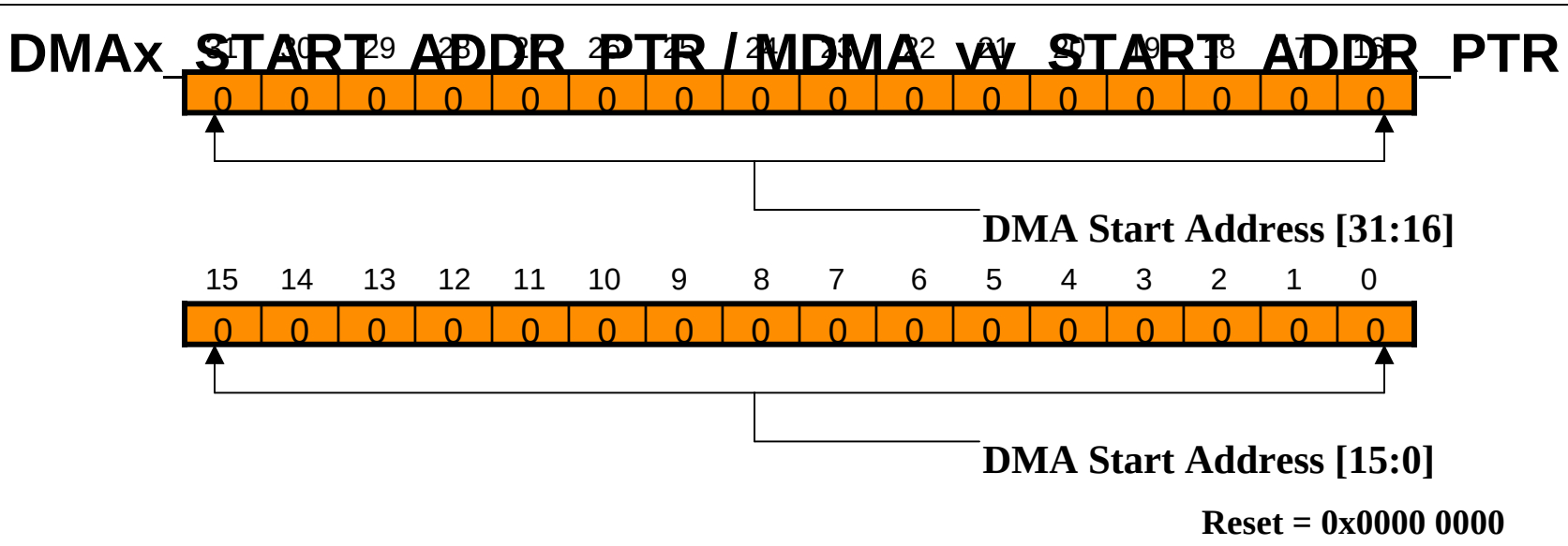
FLOW (Next Operation)
0x0 = Stop
0x1 = Autobuffer Mode
0x4 = Descriptor Array
0x6 = Descriptor List (small model)
0x7 = Descriptor List (large model)

NDSIZE (Next Descriptor Size)
0000 = Required if Stop or Autobuffer Mode
0001 – 1001 = Descriptor Size
1010 – 1111 = Reserved

Interrupt Timing Select
0 = Interrupt after completing whole buffer
1 = Interrupt after completing each row (inner loop), 2D only

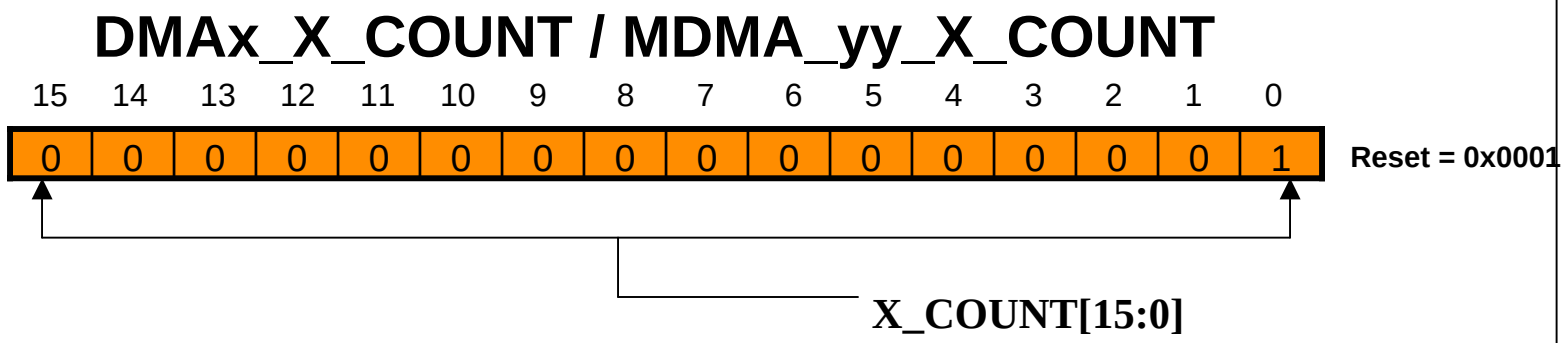
Interrupt Enable
0 = Do not allow completion of DMA transfer to generate an interrupt
1 = Allow completion of DMA transfer to generate an interrupt

Start Address Register



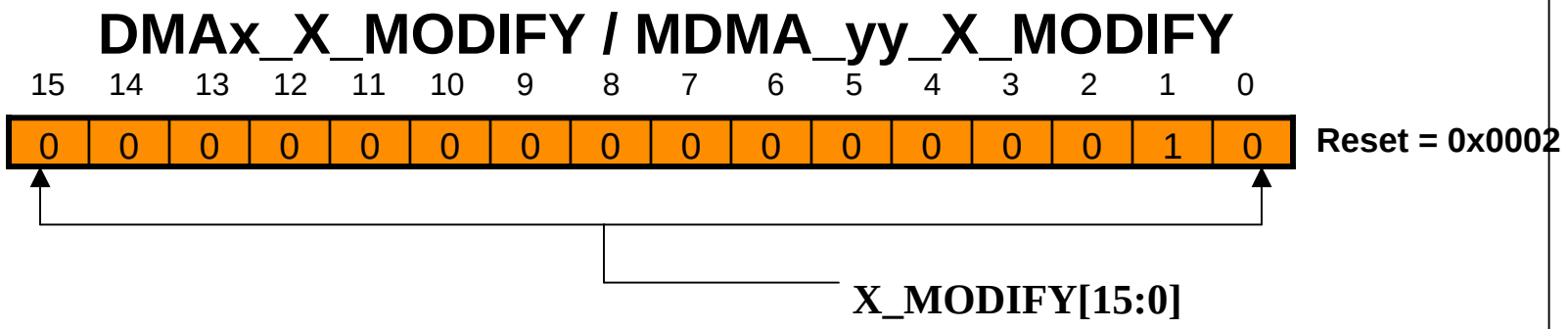
- **Specifies the address of the data buffer currently targeted for DMA. Contents of the Start_Addr_Ptr Register is copied into the Curr_Start_Addr Register at the start of a DMA transfer.**

X Count Register



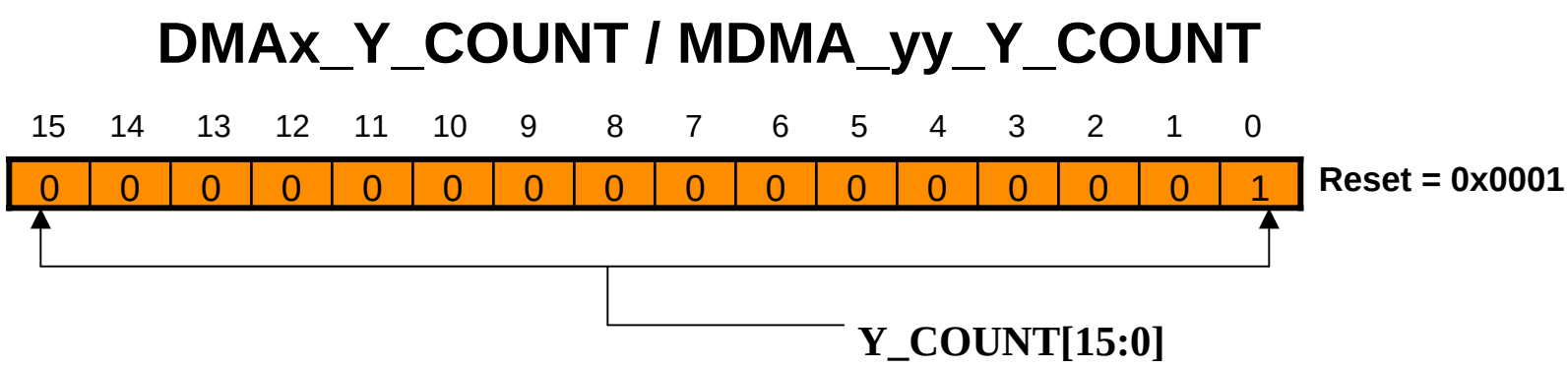
- **For 2D DMA, the X_Count Register contains the inner loop count. For 1D DMA, it specifies the number of elements (8-, 16-, or 32-bit) to read in. A value of 0x0 in X_Count corresponds to 65,536 elements.**

X Address Increment Register



- **This register contains a signed, 2's compliment byte-address increment. In 1D DMA, this increment is the stride that is applied after transferring each element.**
- **In 2D DMA, this increment is applied after transferring each element in the inner loop, up to but not including the last element in each inner loop. After the last element in each inner loop, Y_Modify is applied instead.**

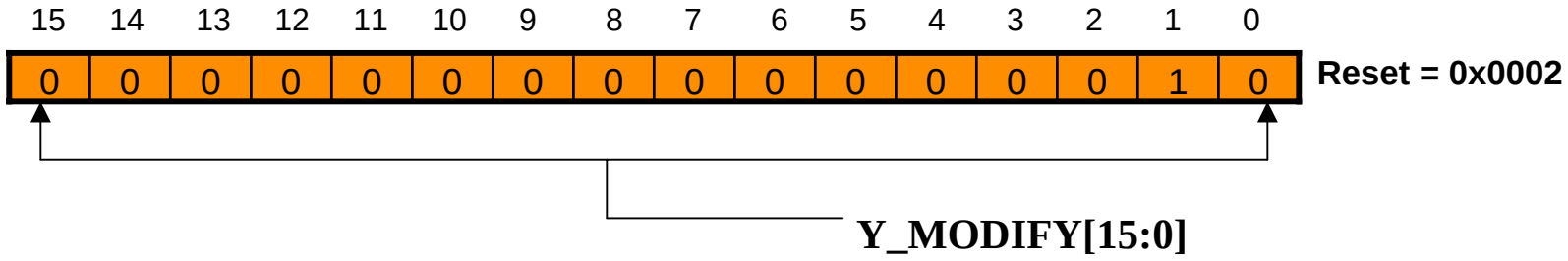
Outer Loop Count Register



- **For 2D DMA, the Y_Count Register contains the outer loop count. This register contains the number of rows in the outer loop of a 2D DMA sequence.**
- **It is not used in 1D DMA.**

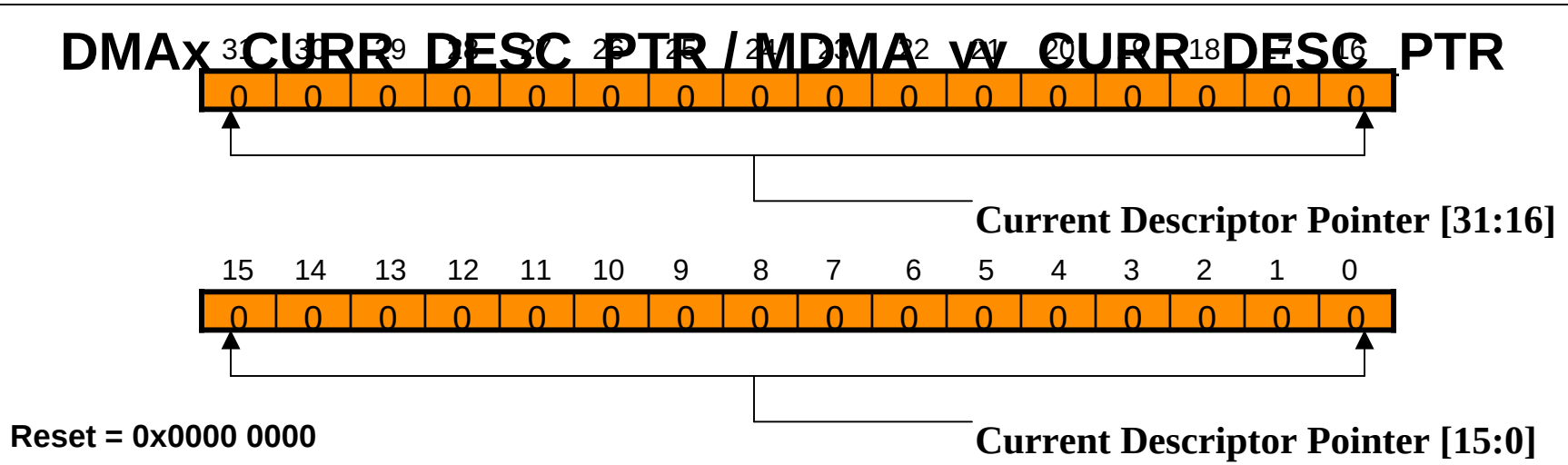
Outer Loop Address Increment Register

DMAx_Y_MODIFY / MDMA_yy_Y_MODIFY



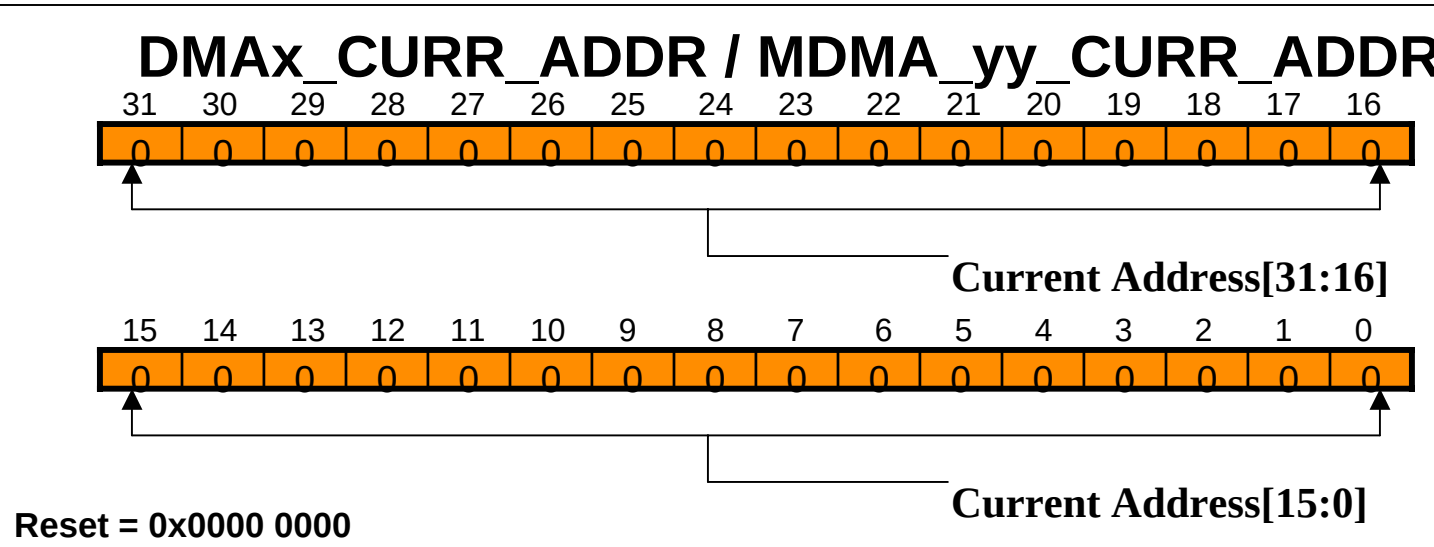
- **This register contains a 2's compliment byte-address increment. In 2D DMA, this increment is applied after each decrement of Curr_Y_Count except for the last item in the 2D array on which the Curr_Y_Count also expires.**
- **The value is the offset between the last word of one "row" and the first word of the next "row"**

Current Descriptor Pointer Register



- **Contains the memory address of the next descriptor element to be loaded. Curr_Desc_Ptr Register increments as each descriptor element is read in. For Descriptor Array Mode, the Curr_Desc_Ptr Register must be programmed, not the Next_Desc_Ptr Register, to initiate a DMA transfer.**

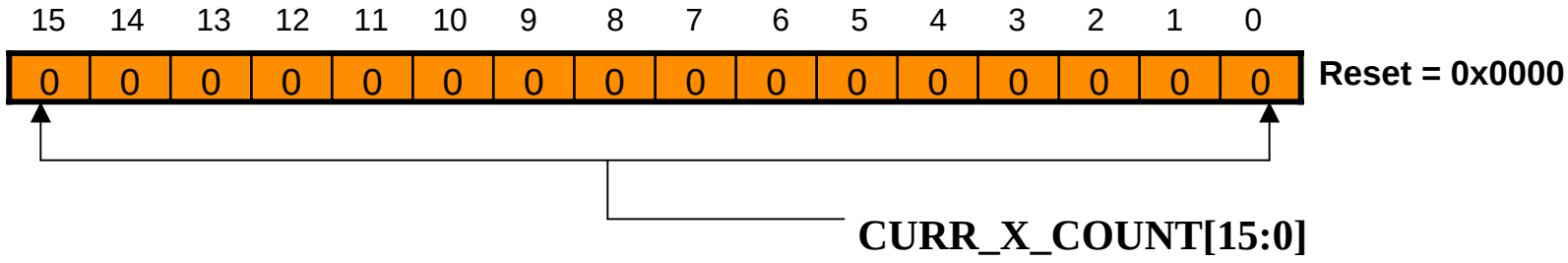
Current Start Register



- **Contains the current DMA transfer address. At the start of a DMA transfer, the Curr_Addr Register is loaded from the Start_Addr Register and it is incremented as each transfer occurs.**

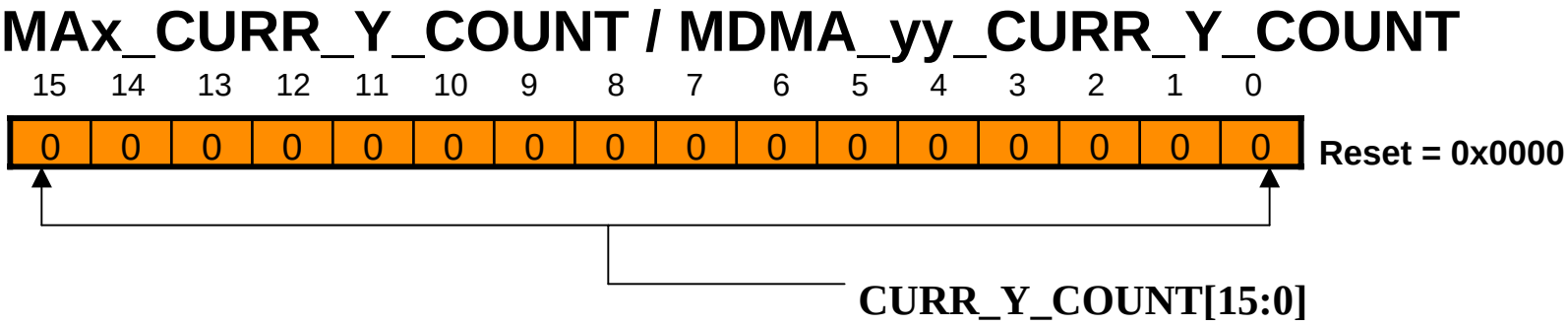
Current X Count Register

DMAx_CURR_X_COUNT / MDMA_yy_CURR_X_COUNT



- **This register is loaded by X_Count at the beginning of each DMA transfer.**
- **It is decremented each time an element is transferred.**
- **For 2D DMA, Curr_X_Count is reloaded after the end of DMA for each row.**
- **Expiration of the count in this register signifies that DMA is complete. In 2D DMA, this register is 0 only when the entire transfer is complete.**

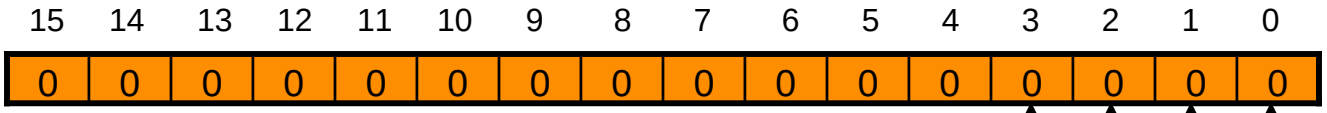
Current Outer Loop Count Register



- This register is loaded by Y_Count at the beginning of each 2D DMA transfer.
- Not used for 1D DMA.
- This register is decremented each time that the Curr_X_Count Register expires during 2D DMA (1 to X_Count or 1 to 0 transition), signifying completion of an entire row transfer.
- After 2D DMA is complete, Curr_Y_Count = 1 and Curr_X_Count = 0

Interrupt Status Register

DMAx_IRQ_STATUS / MDMA_yy_ IRQ_STATUS



Reset = 0x0000

DMA_RUN (DMA Channel Running) – RO
This bit is set to 1 automatically when the DMA_CONFIG register is written
0 – This DMA channel is disabled, or it is enabled but paused
1 – This DMA channel is enabled and operating, either transferring data or fetching a DMA descriptor

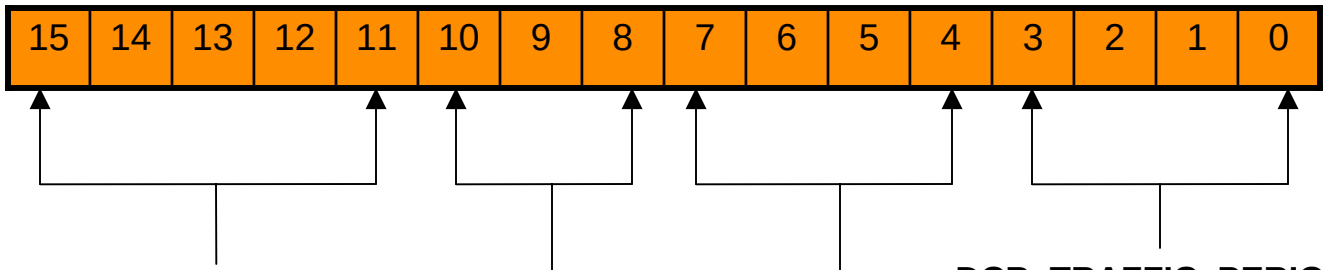
DMA_DONE (DMA Completion Interrupt Status) – W1C
0 – No interrupt is being asserted for this channel
1 – DMA transfer has completed, and this DMA channel's interrupt is being asserted

DFETCH (DMA Descriptor Fetch) – RO
This bit is set to 1 automatically when the DMA_CONFIG register is written with FLOW = 0x4 – 0x7
0 – This DMA channel is disabled, or it is enabled but stopped
1 – This DMA channel is enabled and presently fetching a DMA descriptor

DMA_ERR (DMA Error Interrupt Status) – W1C
0 – No DMA error has occured
1 – A DMA error has occured, and the global DMA error interrupt is being asserted.

DMA Traffic Control Counter Period Register

TC_PER



MDMA_ROUND_ROBIN_PERIOD[4:0]
Max. length of MDMA round-robin bursts. If not zero, any MDMA stream which receives a grant is allowed up to that number of DMA transfers, to the exclusion of the other MDMA streams.

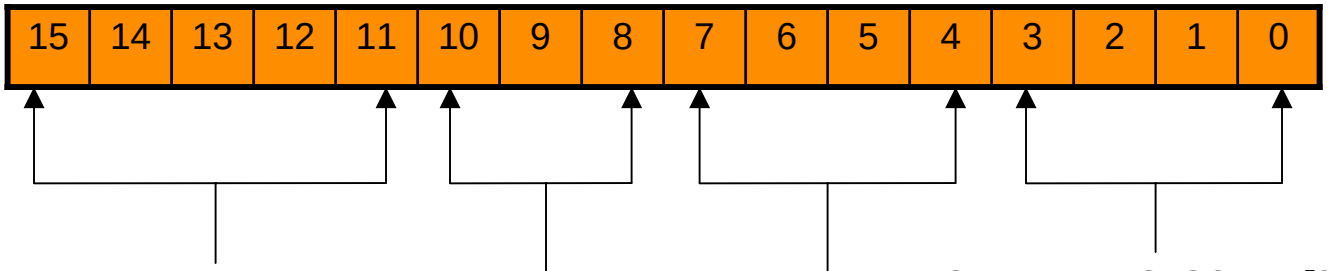
DCB_TRAFFIC_PERIOD[3:0]
000 = No DCB bus transfer grouping performed
Other = Preferred length of unidirectional bursts on the DCB bus between the DMA and internal L1 memory

DAB_TRAFFIC_PERIOD[2:0]
000 = No DAB bus transfer grouping performed
Other = Preferred length of unidirectional bursts on the DAB bus between the DMA and the peripherals.

DEB_TRAFFIC_PERIOD[3:0]
000 = No DEB bus transfer grouping performed
Other = Preferred length of unidirectional bursts on the DEB bus between the DMA and external memory.

DMA Traffic Control Counter Register

TC_CNT - RO



MDMA_ROUND_ROBIN_COUNT[4:0]
Current cycle count remaining in the MDMA round robin period

DCB_TRAFFIC_COUNT[3:0]
Current cycle count remaining in the DCB traffic period

DEB_TRAFFIC_COUNT[3:0]
Current cycle count remaining in the DEB traffic period

DAB_TRAFFIC_COUNT[2:0]
Current cycle count remaining in the DAB traffic period